Computing Reflections

a personal history view

Arthur C. Fleck

Computing Reflections
a personal history view

Arthur C. Fleck
Professor Emeritus
University of Iowa
Iowa City, Iowa

April 2018

Contents

Acknowledgements

Special thanks to Edgar Daylight for his early encouragement to pursue this writing as without that it would not have come to fruition. I am grateful for the careful reading and helpful suggestions provided by Sungwan Kang on earlier versions.

Preface

This treatise sets forth notes and personal impressions on computing from throughout my career. While pursuing my education, there were no computer science departments, and academic opportunities in computing were rare. I indicate the path I followed and how it led me in a seemingly preordained way to a career in computing education.

My career involved both research and teaching, and efforts on both fronts are discussed in the following. I always found a natural attraction for more formal aspects of computing. Initially this was in conceptual models of computing and the universal conclusions they lead to. This is reflected in my early research undertaking in automata theory. But this soon evolved to the more practical concerns of the character of programming languages used to express computations, but still with an analytical orientation involving allied formalisms.

I assess my experience with programming languages extending back 60 years, and cover both well-known and some less well-known languages. Over the years a vast number of programming languages have been conceived so that even being aware of all their names is a challenge (e.g., [143]). My writing involves only a small selective list, and even for those languages that are mentioned, my emphasis is on the context of my experience with the language and related personal digressions rather than comprehensive language coverage. While the sheer number and variation in programming languages can appear daunting, I have come to regard the exploration and evolution of this language diversity as integral to continued advancement in computing, and my comments are intended to express this perspective in various ways.

Arthur Fleck
April 2018

About the author – Arthur Fleck was born in 1936 in Chicago Illinois. He received the Bachelor's degree in Mathematics from Western Michigan University in 1959, and the Master's and Ph.D. degrees in Mathematics from Michigan State University in 1960 and 1964, respectively. After spending one year as an Assistant Professor in the Electrical and Computer Engineering Department at Michigan State, he joined the Computer Science Department at the University of Iowa. Except for a sabbatical year at the University of Virginia, he remained at the University of Iowa. He served as department chairman there for two terms from 1984 to 1990, and retired in 2006.

Chapter I
My Initiation into Computing

A Serendipitous Choice

I first encountered a computer in 1956. I was a junior majoring in mathematics at Western Michigan University and I decided to enroll in a new computer programming course taught by Mr. Jack Meager. The computer used for the course was an IBM 650, the earliest mass-produced computer with nearly 2000 produced between 1954 and 1962. The IBM 650 (see Figure 1) used 80-column punched cards for input and output, and internal storage was on a rotating magnetic drum organized into words consisting of ten decimal (technically bi-quinary) digits. The particular computer our class used had 1000 words of memory (a 2000 word memory was an option).

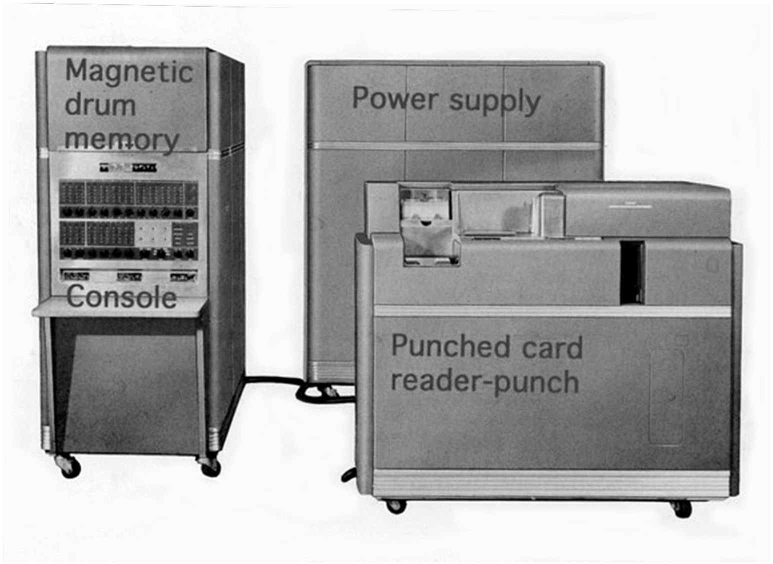

Figure 1: The IBM 650 computer (source: Fig. 1 [81])

A considerable complication for our class was that at the time, Western Michigan University had no computer at all available for student use! Fortunately this class was small (under ten), and our enterprising instructor arranged for the class to make after-hours use of the data processing center at Whirlpool Corporation some 50 miles away. It was amazingly generous (and brave) of a major corporation to turn a group of undergraduates loose in their data center, but it was a fantastic opportunity for us. So twice per week, the class spent a long evening traveling, using punch card equipment to prepare, and sitting at the console debugging programs conceived during the intervening time. Since the opportunities to “try it” were so few, this extremely limited access to a computer instilled a lasting appreciation for meticulous care during program creation. The only ‘text’ I recall for this course was the machine manual [81].

The use of rotating magnetic drum storage was a predominant feature of the architecture of the IBM 650. The drum surface was divided into bands that circle the drum and each band contained 50 words. A read/write head was provided for each band (20 on the computer we used). The drum rotated at 12,500 revolutions per minute, so one revolution required 4.8 milliseconds, the maximum delay to access a storage location. This created the possibility of a great variation in the execution time of an individual instruction. For example, under optimal conditions an add instruction (fixed point, there was no hardware floating point) would take .768 milliseconds. But depending on the placement of the data and the instruction, in the worst case it could take two revolutions to access first the instruction, and then its data from the address in the instruction. This made using sequential storage locations of instructions and data inherently inefficient. As a consequence, the IBM 650 adopted a two-address instruction format where the first address was for the data used by the instruction, and the second was the address of the instruction to be executed next. This allowed complete freedom for the choice of location of every instruction and data value.

A detailed instruction timing calculation could determine an optimal drum location for each data item and each instruction ensuring their locations would reach a drum read-head with minimum delay after access was required. This could provide up to ten-fold improvement in the execution efficiency of a program over sequentially located instructions and data. A “by hand” timing analysis to determine optimal location placement was an extremely tedious additional step once the basic program was written. IBM provided the Symbolic Optimal Assembly Program (SOAP) to automate (an approximation to) this task. Our instructor briefly discussed SOAP in class, but it was not available to us at the facility we used. The greatest gain from location optimization was in critical program loops, but in our class work we routinely ignored optimization considerations. I recall only once that this was pursued on a small inner loop as an exercise, but that did provide an early focus on a basic principle for program optimization. For class projects we coded directly in machine language, and there was no operating system to provide supporting services.

Machine (and assembly) language is so linguistically primitive that I question referring to it as a “language”. Assembly language only provides symbolic reference to op-codes and memory addresses rather than machine numeric, and it includes almost no linguistic structure. Better might be calling these “notations” rather than languages, but that’s not the usual convention so I won’t protest any further. But issues concerning the linguistic characteristics of programming languages become a major topic later.

Western Michigan University had just this one course in computing at this time. The course was offered in the Mathematics Department (it would be ten years before Computer Science Departments began to emerge at U.S. universities). As a math major I found this to be one of the most interesting courses I took. I did repeat the course one additional time (at the instructor’s invitation, but for no credit) and gained some further experience, but there were no other locally available opportunities at that time. Despite its brevity, I found this an informative, even compelling, initiation into computing. However at that point it seemed like an interesting diversion not at all like a career direction. But the solid mathematics background I received would serve me well when later events led in that direction.

A noteworthy coincidence occurred during my senior year at Western Michigan University. I was invited to join an honorary science society, and the induction ceremony revolved around a dinner. The after-dinner speaker at that function was Dr. Gerard Weeg, a mathematics professor and Computer Laboratory staff member at Michigan State University. He was a fantastic after-dinner speaker, brilliantly mixing humor with an intriguing discussion of computer research projects taking place in their Computer Lab. It was fortuitous that I had some background relevant to technical issues in this fascinating talk that afforded me a deeper appreciation of its content. Although I had no idea at that moment that either Michigan State or Dr. Weeg would play any role in my future, that event turned out to be a harbinger of my future.

Getting Serious

As I neared the completion of my undergraduate degree I was considering a job offer in industry. But I belatedly decided to pursue a graduate degree. I applied for admission to several graduate mathematics programs, including teaching assistantship applications. My admission applications were successful, but the teaching assistantship applications were not. Because of an advising glitch I had to spend an extra semester and so I finished my undergraduate degree in mid-year, and the middle of the academic year proved to be an unfortunate time to seek a teaching assistantship. Since we had already started a family, financial assistance was a necessity, and the way forward looked uncertain. However, it was my good fortune that the Head of the Mathematics Department at Western Michigan University, Dr. James Powell, learned through personal contacts of the availability of a research assistantship in the Computer Laboratory at Michigan State University.

So I submitted an application to the Computer Laboratory at Michigan State University (MSU) and as fate would have it, in January of 1959 I entered the graduate mathematics program at MSU, and began a research assistantship in the Computer Laboratory under the direction of Professor Weeg! The confluence of a number of apparently unrelated factors had combined to lead me to one of a relatively few places at this time where the pursuit of computing knowledge and education was the central concern.

The Computer Laboratory was the computing service unit of MSU. A computer named MISTIC (see Figure 2) had been built under the direction of Dr. Lawrence Von Tersch, and became operational in late 1957. Dr. Von Tersch was both the founder and the current Director of the Computer Lab. MISTIC was a duplicate of ILLIAC I at the University of Illinois, and was the first computer on the MSU campus. MISTIC was a binary machine with 40-bit words and 1024 words of internal vacuum tube memory. This machine was quite fast for its time – under 100 microseconds for an ‘add’ operation (fixed point, there was no hardware floating point). Input-output was via paper tape and teletype equipment. Over the years of its operation, staff at the Computer Laboratory eventually constructed and added an expanded magnetic core memory, and connected punched card equipment for input-output.

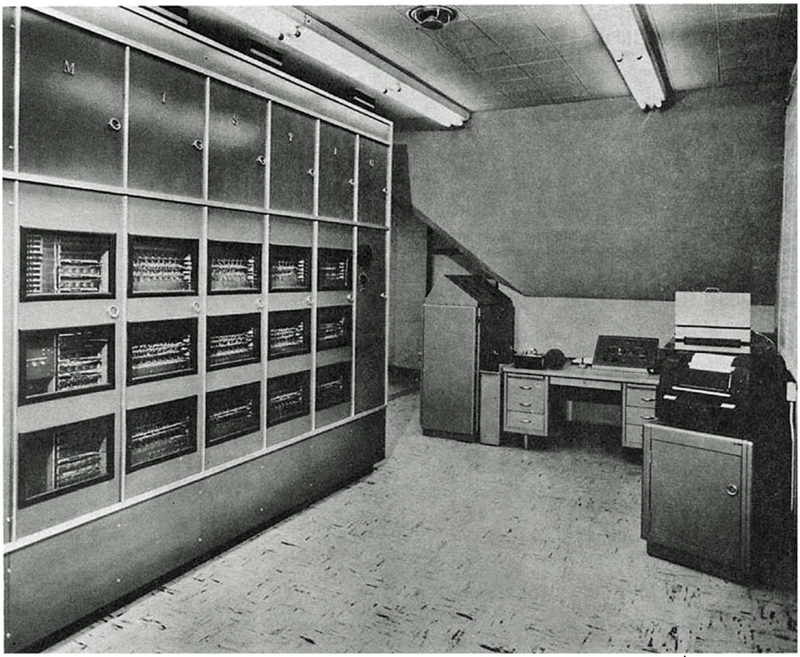

Figure 2: MISTIC computer, constructed in 1957[1]

My initial duties in the Computer Lab were programming assignments for MISTIC, the sole computer at the Computer Lab for several years after my arrival. But in the early 1960s the Lab was anticipating retiring MISTIC through the acquisition of commercial computers from the Control Data Corporation (CDC). The first arrival was the CDC 160A, a small (for the day) desk-size computer, followed a year or so later by the CDC 3600, a powerful mainframe computer. These machines had architectures that contrasted sharply with that of the IBM 650 and MISTIC. The CDC 160A and 3600 were “modern” computers with index registers, asynchronous I/O, interrupt driven processing, magnetic tape and high-speed peripherals. The assembly language of each of these machines bore the mark of its architectural character.

[1] Photo courtesy of Michigan State University Archives

My programming work during this period was largely on systems programming tasks. In fact, the 160A and 3600 could be cabled together to form an early commercial multi-processor system, and one of my later programming projects employed that configuration. Several long-term users of the Computer Lab had substantial investment in software for MISTIC and were deeply concerned about its impending retirement. To dispel these concerns I was assigned to construct a simulator for MISTIC on the new computer(s). The simulator was written (in assembly language) for the CDC 3600, but the 3600 did not have a paper tape reader, and as a primary input device of MISTIC, paper tape reader operation was embedded in much of its software. However, the CDC 160A did have a paper tape reader, so the coupled computers were used to program a simulator that provided unimpaired execution of any running MISTIC program. This provided early (about 1962-63) experience with multi-processor programming.

The extensive hands-on experience with the MISTIC computer was a real touchstone. As a copy of the ILLIAC I computer constructed at the University of Illinois in 1952 [119], MISTIC shared the heritage of being directly based on the historic computer architecture devised by John von Neumann and others in 1945 [136]. Having only the most basic facilities of a von Neumann computer, programming on MISTIC routinely required techniques of program self-modification. A number of alternative features to replace that need were already included in the computers replacing it. Computer architecture advances such as interrupt driven facilities, asynchronous input-output, and various addressing modes would lead to rapid evolution of programming techniques. Even back then, the MISTIC experience felt like an encounter with computing history.

Chapter II
A History of Automaton Automorphisms

The Excitement of New Ideas

In addition to my programming responsibilities in the Computer Lab at MSU, I immediately joined an on-going reading group of research assistants led by Professor Weeg. This group read and met regularly to discuss theoretical work in computing. We studied the historic work on Turing computability using the book by Martin Davis [29] as the source. But much emphasis was given to the finite state machine model that came to the forefront in the mid-1950s. We studied papers such as Mealy [106], Moore [107], Ginsberg [63], and Rabin & Scott [116] as well as others that began to appear. This model was developed to aid in the design of complex circuits following primary contributions by Huffman [79] and Kleene [91]. Inspiration for this formal model can be traced back to Shannon's work on relay circuits in 1938 [123], and McCulloch & Pitts's work on nerve-nets in 1943 [105]. But the papers published in the mid-1950s established the finite state model as it has come to be known. This background provided me with crucial early preparation and direction for eventual thesis research. However, it would remain unclear for some time how, or if, these topics might relate to a dissertation in traditional mathematics (the Computer Science Department at Michigan State University did not commence operation until 1969).

In 1960 Professor Weeg guided me in the initiation of research activities and asked about my preferences. After a bit of consideration, I was motivated by studies in topology where ideas of structure preserving (e.g., continuous) transformations are captured in a very general, abstract and axiomatic way. So I suggested using an analogous approach to investigate transformations on automata (e.g., reduction to a minimal state machine), although I had some doubt this suggestion would be well received. Earlier Professor Weeg had worked on algorithms for floating point arithmetic for the UNIVAC 1103 at Remington Rand, and at this time he was deeply involved in research and writing for a book on numerical analysis [142]. So it was unclear to me if he would welcome my striking off in a new and unestablished direction. But I was pleased (and relieved) that Professor Weeg gave his approval. A disadvantage of this research direction was that there was no preceding work at all to offer guidance on this topic. But this also allowed me to dispense with the usual first research step of learning what had already been done. Almost at once we would be in the midst of the creation of an entirely new area of automata research.

Our arrangement was that I would meet with Professor Weeg once a week to discuss with him whatever research progress I had made. The topological analogy that I chose for automata was to use submachines as the open sets. This did lead to a continuous function concept with desirable properties for the preservation of numerous automaton structures of interest. However, in most cases the topology was uninteresting, and in cases of the most interest (strongly connected automata), the topology was trivial. So while it did provide initial results of interest, this avenue of research was soon exhausted. But making original discoveries was exciting, and seeking to continue to pursue investigations of automaton transformations led me to consider following analogies with ideas in algebra.

The automaton transformation functions I next considered were inspired by the homomorphism concept from algebra, and for automata transformations I dubbed them “operation preserving”. The operation to be preserved is the state transition function that embodies the structure of an automaton. So this might alternatively be called a transition preserving transformation. The term I used appeared in a number of early published papers. Because of a clear analogy with mappings in algebra, as work progressed the term *homomorphism* would sometimes replace it. These transformations have strong structure preserving properties. The idea is perhaps most simply communicated with a diagram. In Figure 3, we focus on a single transition between two states in an automaton A, and the condition that is required of an operation preserving function h in the image automaton B, where h: A → B.

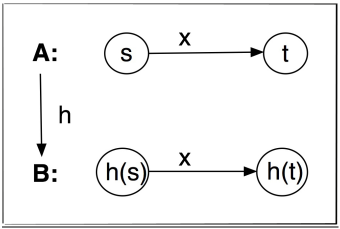

Figure 3: operation preserving function

In Figure 3, target states h(s) and h(t) in automaton B are required to enjoy exactly corresponding transitions to each of those in their source states s and t in automaton A. This is often referred to as a "commuting diagram" as when you follow h and then x in B you always arrive at the same state as when you follow x in A and then h. Basic details of the definitions will be more fully described in the next section.

My weekly meetings with Professor Weeg continued as I pursued this new direction. But there soon was a change in our interaction. In addition to me reporting my new results, Professor Weeg began informing me of new results he was obtaining on this topic. This was quite an unexpected turn of events to me because of his focused activity in numerical analysis as noted earlier. Of course, he had been leading our research seminar on automata theory, so a primary interest there was clear. But he had not previously published in this area. Only later did I gain a fuller appreciation when I learned that Professor Weeg's doctoral dissertation was in algebra. At any rate, the nature of our interactions shifted to a mutual exchange from that point on.

With Professor Weeg's guidance I completed research for a dissertation to submit to the Mathematics Department just as he took a sabbatical leave (for the 1963-64 academic year to Iowa State University where he had done his Ph.D.). I had finished writing my thesis before his departure, but a copy did not get finalized and sent to my committee until the start of that academic year. There ensued a long delay with no reaction and no setting of a defense date. Professor Weeg's absence complicated communication both with me and evidently with my committee members. Eventually the defense delay extended through the entire academic year. Only after my defense did I learn that this was due to one committee member, Dr. John Hocking. Dr. Hocking was a topologist, and as I indicated earlier, the level of topological ideas in my initial approach to investigating automata mappings was superficial. Apparently this (and I surmise perhaps the non-traditional mathematical topic of my work) left him reluctant to accept the thesis. This was ironic as my inspiration for this perspective on automata transformations took root from a topology course I took from Dr. Hocking.

Despite the lack of depth of topological ideas in my initial work, the automata mapping results arising from the topological perspective were significant and original from the automata point of view. And in any case this was a small minority of the thesis with the vast majority of work pursuing the algebraic line that was my primary focus. It appeared that there might also have been political issues. Professor Weeg was only part-time in the Mathematics department, had his primary office at the Computer Lab, and had research interests primarily related to his Computer Lab activity. In addition, as I found out considerably later, Professor Weeg had determined by early that spring that he would not return to MSU after completing his sabbatical as normally required, and there was a dispute over the repayment of the partial financial support he received from MSU during his sabbatical. At any rate, late in the spring quarter a date was finally set for my defense. I found out after my defense that Dr. Hocking had agreed to accept the judgment of an unbiased, external scholar. This was Seymour Ginsburg who Dr. Hocking knew from their days together in the graduate mathematics program at the University of Michigan. Ginsburg had a well-established reputation as an automata theory scholar by this time (a reference to some of his work was included at the beginning of this chapter), and subsequently enjoyed a long and distinguished career in computer science. Happily, Ginsburg's opinion of my dissertation was reportedly completely positive, and my thesis defense itself was routine.

The remainder of this chapter will set forth the origin of the automaton automorphism topic in some detail, including a selection of early results. Since proofs of all the results already appear in the literature, they are omitted, except when those ideas are crucial to explaining the direction of the research.

Basic Definitions

I first present the model adopted for our research. Although there has been little or no real change in the concept over the years, the notation to be used has evolved. Only a selected bibliography is provided in the following, but the book by Bavel [14] includes substantial bibliographic coverage of the topic.

Definition 1 [36, 38]: an **automaton** A is a triple, A = (S, I, M), where S is a finite set (the *states*), I is a non-empty semigroup (the collection of *inputs*), and M: S × I → S is a function (the *state transition function*). We assume the "sequential machine" condition that M(M(s, x), y) = M(s, xy) for all x,y∈I and s∈S (the juxtaposition xy denotes the application of the semigroup operation to elements x and y).

Several remarks are relevant to this definition. First of all, the symbols used to denote the set of states, the inputs, and the transition function are arbitrary and can be chosen to suit personal taste. The popular convention for the choice of symbols has varied over the history of the topic, in part guided by the evolution of commonly available typography. It is written here as in the first published papers on automaton automorphisms, and this continuity will be followed throughout. Definition 1 (mostly) followed the notation adopted by Rabin and Scott [116]. However, this is both a departure from, and a generalization of, the definition adopted by most authors at this time, including Rabin and Scott. The assumption that the state set S is finite is non-essential for many results and was not included in our early work.

But it is assumed by numerous other authors to be cited and so is adopted for consistency (and, of course, inspired by the concept of the “finite state model”).

Definition 1 is a departure from most other work at this time by the omission of both outputs of a transducer and final states of a recognizer. Some authors subsequently referred to this model as a *semi-automaton*, or quasi-automaton, acknowledging that something is missing from an “automaton”. Other authors have preferred the term *transition system*, perhaps the most fitting choice. But as others have suggested, the term “automaton” is quite pliable, and it is the term we (and many others) have used so it will be continued here. This model is concerned only with state change in response to input, and that is a central point as this research intends to focus on aspects of the internal structure of the state change of automata.

The generalization aspect of Definition 1 is allowing an arbitrary semigroup to serve as inputs. The semigroup aspect is an abstraction allowing a wide variety of “input elements”, and only requires of them a binary operation that is *associative*. Ginsberg [64] first introduced this generalization. Most authors at that time (and many now) assumed instead that the inputs consist of all finite sequences, written Σ^*, where Σ is a finite alphabet of symbols (similarly for outputs with their own alphabet when they are included). But that case is subsumed in Definition 1 as Σ^* is a semigroup (monoid in fact) under the concatenation of sequences, and the flexibility gained (e.g., modeling gesture input) by the generalization in Definition 1 rarely adds complication.

Definition 2 [36, 38]: Given two automata $A_i = (S_i, I, M_i)$, $i = 1, 2$, a function $h: A_1 \rightarrow A_2$ is understood to be a function from S_1 into S_2; the function h is said to be **operation preserving** if $h(M_1(s, x)) = M_2(h(s), x)$ for all $s \in S_1$ and $x \in I$.

The operation preserving property is the formalization of the condition depicted in Figure 3 (commuting of the applications of function h with applications of inputs). If the function h is also one-to-one and onto, it is called an **isomorphism**; when $A_1 = A_2$, the function is called an **endomorphism**, and if also an isomorphism, it is then referred to as an **automorphism**.

Other authors who study this type of transformation sometimes allow the two automata to have different inputs and include a mapping between the inputs as well. We have discarded that generalization since the focus is on the structure of transformations within a single automaton.

The first writings to appear on automaton automorphisms were individual technical reports by Professor Weeg and myself [137, 36] in May of 1961. During this timeframe, the Computer Lab at MSU and many computing research organizations produced a technical report series, and they were quite widely exchanged between institutions. So while not a "published" source, citations to these technical reports appeared in a number of early publications on the topic. The first "external" presentation of the ideas was my talk at the summer meeting of the American Mathematical Society in August of 1961 [37]. The first publications were separate journal articles by Professor Weeg and myself [138, 38] and a conference paper by Professor Weeg [139], all in 1962 (although the conference paper did not appear in print until 1963). Also, my dissertation [39] was eventually accepted by the Mathematics Department in 1964. I next will highlight the major results in these sources.

Initial Results

One algebraic structure naturally associated with an automaton, the characteristic semigroup, derives from the interplay between input semigroup characteristics and the automaton structure.

Definition 3 [36, 38]: inputs $x,y \in I$ of an automaton $A = (S, I, M)$ are **equivalent** (with respect to A), $x \equiv y$, provided that $M(s, x) = M(s, y)$ for all $s \in S$.

The equivalence classes from this relation partition the inputs I into disjoint sets that are written as $[x] = \{y \in I \mid x \equiv y\}$. These classes themselves form the **characteristic semigroup**, S(A), under the natural (and unambiguous) operation $[x] \bullet [y] = [xy]$, and we normally just write this operation as the juxtaposition $[x][y]$. S(A) is often referred to as a *quotient* of I by $\equiv$ and written as $S(A) = I/\equiv$. The natural correspondence $h\colon I \to S(A)$ given by $h(x) = [x]$ is a semigroup homomorphism.

For automaton $A = (S, I, M)$, each input $x \in I$ defines a *state function* $M_x\colon S \to S$, which we write here with postfix notation, by $(s)M_x = M(s, x)$. Of course, another input $y \in I$ may define the same function, $M_x = M_y$ as functions. But this is true if and only if x and y are equivalent ($x \equiv y$), so M_x is just another name for the equivalence class $[x]$. And $((s)M_x)M_y = (s)M_{xy}$, so the (finite) collection $\{M_x \mid x \in I\}$ is a functional representation isomorphic to the characteristic semigroup S(A).

A large variety of properties of the internal structure of automata have potential interest, but we will confine this presentation to (effectively) four as we proceed.

Definition 4 [115]: an automaton $A = (S, I, M)$ is **cyclic** if there exists a state $s_0 \in S$ so that for each $s \in S$ there exists $x \in I$ with $M(s_0, x) = s$; the state s_0 is called a **generator**; automaton A is **strongly connected** if every state is a generator (so that transitions exist between each pair of states).

Although implicit in work such as Rabin and Scott's [116], the idea of a cyclic automaton was first explicitly identified by Oehmke and was named for an analogy from algebra. But this analogy has a significant flaw that will be revealed shortly. Oehmke introduced the cyclic idea a little after the time period under discussion in this section. Later on Bavel introduced [12] the name **singly generated** automata to replace the cyclic automata terminology, and I find this a much better choice for the reason to be indicated shortly. However, it is most succinct and natural to include this concept at this point, in conjunction with its global counterpart, strongly connected automata (the idea of immediate historical relevance). Strongly connected automata were introduced much earlier by Moore [107] in his study of the input-output distinguishability of states in transducers.

It is readily verified that if an operation preserving function maps a singly generated or strongly connected automaton onto another automaton, then the image automaton is also, respectively, singly generated or strongly connected.

Groups are a thoroughly studied class of algebraic systems and one of the earliest of my observations prompted anticipation that some of this understanding may lead to insight into automaton structure.

Theorem 1 [36, 38]: the collection of automorphisms of an automaton A forms a group. The symbolism chosen for this **automorphism group** was G(A).

By similar analysis, but lacking inverses, there is an **endomorphism semigroup**, denoted by E(A), formed by that collection. We will explore numerous relationships between S(A), G(A), E(A), and the structure of A. This commences with an early result that was a seemingly small observation but with substantial consequences.

Theorem 2 [36, 38]: given two automata $A_i = (S_i, \mathrm{I}, M_i)$, i = 1, 2, an operation preserving function $h: A_1 \rightarrow A_2$, and a generator state s_0 of A_1, h is completely determined by its value for $h(s_0)$.

Theorem 2 follows since for any state $s \in S_1$ there is $x \in \mathrm{I}$ with $M_1(s_0, x) = s$, but then $h(s) = h(M_1(s_0, x)) = M_2(h(s_0), x)$. Of course, we don't expect an arbitrary choice of $h(s_0)$ to necessarily be consistent with an operation preserving function (e.g., since generator states must map to generators), so valid options require verification. But for operation preserving functions, an important corollary in the case of strongly connected automata is that two *different* operation preserving functions h_1 and h_2 ($h_1 \neq h_2$) must produce different results for *every* state argument (i.e., if there exists a state s with $h_1(s) = h_2(s)$, then $h_1 = h_2$).

This unique determination of an entire function from its action on a single argument greatly simplifies the discovery of operation preserving functions. In particular, we can conclude [36, 38] that the size of the group G(A) cannot exceed the number of states in A if A is *singly generated* – briefly stated $\#G(A) \leq \#S_A$ (we are using #X to denote the cardinality of set X). However, while this proof was given in the first publications, this result was only asserted for strongly connected automata as the singly generated concept had not yet been proposed (in fact, the same proof also establishes that $\#E(A) \leq \#S_A$ for singly generated A).

Professor Weeg had discovered that the automorphisms of strongly connected automata have rather special properties. An automorphism acts as a permutation of states. A **regular permutation** (on a finite set) is one that is the product of disjoint cycles all of the same length.

Theorem 3 [137, 139]: if A = (S, I, M) is a strongly connected automaton, then G(A) is a group of regular permutations, furthermore each group of regular permutations occurs as G(A) for such an automaton.

Then from known properties of regular permutations:
Corollary [137, 139]: the order of G(A), #G(A), is an integral divisor of #S.

This provides a definitive structure for automorphisms and improves the earlier result that $\#G(A) \leq \#S$. Professor Weeg [139] also provided an algorithm to determine G(A), and conversely a constructive method to develop a strongly connected automaton A for which G(A) is a given group of regular permutations.

Professor Weeg explored a sense in which the structure of G(A) is reflected by the behavior of classes of inputs.

Definition 5 [137, 138]: for automaton $A = (S, I, M)$ and $s \in S$, inputs $x,y \in I$ are **in agreement** (with respect to s), $x \sim_s y$, if and only if $M(s, x) = M(s, y)$. This is an equivalence relation that partitions inputs into classes written $[x]_s = \{y \mid x \sim_s y\}$. Also, for each pair of states $s,t \in S$ there is a set of associated *transition inputs* $I_{st} = \{x \in I \mid M(s, x) = t\}$.

Each agreement equivalence class may merge several of the equivalence classes of the characteristic semigroup. Since the agreement classes depend only on a single state they are more readily determined than equivalence classes. Weeg considers a partition of each agreement class $[x]_s$ as a union of transition sets, $[x]_s = I_{st} \cup I_{su} \cup \dots$, where the union extends over the transition sets I_{st} for each $t \in S$. Furthermore, the transition sets I_{st} and I_{su} are disjoint when $t \neq u$, and all are non-empty if A is strongly connected.

Finally, Weeg defines an operation ($*$) on the transition sets by $I_{st} * I_{su} = I_{sv}$ provided that $M(s, xy) = v$ for all $x \in I_{st}$ and $y \in I_{su}$ and he obtains a result connecting automorphisms to this transition set operation.

Theorem 4 [138]: if $A = (S, I, M)$ is a strongly connected automaton and G'(A) is a subgroup of G(A), then for each $s \in S$ there is a collection of transition sets contained in $[x]_s$ which under the $*$ operation is a group isomorphic to G'(A).

We next take up a structure property arising from an observation about the relation between endomorphisms and state transition behavior. Suppose there is an input $x \in I$ and an endomorphism h that "track" one another, i.e., $M(s, x) = h(s)$ for all $s \in S$ (or $M_x = h$). Then since h commutes with all state transitions, the same must be true of transitions by x. That is, for all $y \in I$ and all $s \in S$, $M(s, xy) = M(M(s, x), y) = M(h(s), y) = h(M(s, y)) = M(M(s, y), x) = M(s, yx)$. Whether an input and endomorphism exist that enjoy this relationship is in question, and this leads to the following idea.

Definition 6 [36, 38]: an automaton A = (S, I, M) is **abelian** provided M(s, xy) = M(s, yx) for all s∈S and x,y∈I; if A is abelian and strongly connected, it is called **perfect**.

Here is where we encounter the flaw in the choice of using the algebra analogy in Oehmke's selection of the name *cyclic* automaton. In algebra, a cyclic group (or semigroup) is necessarily an abelian group (resp. semigroup). However, for automata, the cyclic property *does not imply* the abelian property (it is easy to provide even a two state counter-example). Hence I find it apposite to avoid that potentially misleading 'cyclic' term and use 'singly generated' automata instead.

Any automaton with an abelian input semigroup is abelian, but abelian automata may readily occur with an arbitrary input semigroup. We also note in passing that the abelian property of automata is also invariant under operation preserving functions.

In algebra, the abelian case of systems commonly has a simpler analysis than the general case. As we will see, the same circumstance obtains for automata. For an abelian automaton, each function M_x is operation preserving since its application commutes with every input. That is, every input defines an endomorphism. Further exploration in this direction leads to the next result.

Theorem 5 [36, 38]: If A = (S, I, M) is a perfect automaton, then G(A) is an abelian group, and for each automorphism α, $\alpha = M_x$ for some x∈I.

This result characterizes automorphisms in terms of input behavior and that leads to even further conclusions for perfect automata.

Theorem 6 [38]: If A = (S, I, M) is a perfect automaton, then #G(A) = #S and G(A) = S(A) (i.e., every endomorphism is an automorphism and their number is the same as the state count).

Professor Weeg also proved [138] the converse of Theorem 6 – i.e., if A is strongly connected, then A is abelian if and only if G(A) is abelian and #G(A) = #S.

To conclude this line of investigation we introduce our last structure concept of interest.

Definition 7 [116]: Given automata $A_i = (S_i, I, M_i)$, i = 1, 2, their **direct product** is $A_1 \times A_2 = (S_1 \times S_2, I, M_1 \times M_2)$, where $M_1 \times M_2((s_1,s_2), y) = (M_1(s_1,y), M_2(s_2, y))$, for all $y \in I$, $s_1 \in S_1$, and $s_2 \in S_2$.

The direct product was introduced by Rabin and Scott as a means to construct a recognizer for the intersection of regular languages from their individual recognizers. Actually, it provides even more since the one direct product automaton can recognize not just the intersection, but also each of the two individual languages, their compliments, and their union, just by choice of a set of final states. But this only scratches the surface of this idea. The direct product provides a basis for the study of the decomposition of automata into smaller and/or simpler components. It is with that perspective that we conclude this section.

Theorem 7 [38]: if A is a perfect automaton, then A is a direct product of automata if and only if G(A) is a direct product of groups.

Since factor automata must again be perfect, Theorem 7 can be applied repeatedly. From group theory it is known that an abelian group is a direct product of cyclic groups of prime power order, so we conclude that perfect automata must decompose into exactly such components (i.e., each input acts as a cyclic transition of prime power order that is a factor of the original state set size). Hence in this tightly restricted class of perfect automata, we see a precursor to the fully general decomposition theory subsequently developed by Krohn and Rhodes [95, 96].

We note in passing that chapter 5 of Gécseg & Peák [61] provides a redevelopment incorporating some of the results summarized in this section and the next, includes additions of their own, and provides a bibliography that includes Russian and eastern European work not reported on in this document.

Ensuing Developments

Interest in the automaton automorphism topic spread quite rapidly. First locally, Robert Oehmke, a colleague of Professor Weeg in Mathematics, became interested in the area and published an early paper [115] that has already been cited. Also within the first year, two more of Professor Weeg's doctoral students began work in the area. This was Ralph Brown and Bruce Barnes, and before long they published results [19, 11]. This together with additional papers by Professor Weeg and myself [140, 141, 40, 41] provided a critical mass of research that attracted a national and international collection of researchers to the area. The sense of the ensuing follow-on effort is presented here by discussing a selection of recurrent threads in the work.

Our first thread encompasses transition analysis and the characteristic semigroup. Weeg [140] refined his earlier work on agreement classes for strongly connected automata by exhibiting a homomorphism from a subset of the inputs I to G(A). Brown [19] and Barnes [11] continued the earlier investigation by Weeg [138]. Brown showed that Weeg's basic results for strongly connected automata also hold for singly generated automata (although he used that condition without naming it) as long as consideration is restricted to agreement classes $[x]_s$ where s is a generator.

Barnes [11] continued the work from [138] by showing that for a strongly connected automaton, if the complete collection of transition sets of $[x]_s$ for a state forms a group, then G(A) is isomorphic to that group. He also established an interesting relationship between subgroups of an automorphism group and subautomata (where an automaton (S', I', M') is a *subautomaton* of (S, I, M) if $S' \subseteq S$, I' is a subsemigroup of I, and M' is M restricted to $S' \times I'$).

Theorem 8 [11]: for each strongly connected automaton A and subgroup G'(A) of G(A) there is a strongly connected subautomaton A' whose automorphism group G(A') is isomorphic to G'(A).

Later Uemura [134] used semigroup methods to show conditions for G(A) to be (isomorphically) embedded in S(A), and Masunaga, Noguchi and Oizume [103] presented an incisive unification of numerous variants of automata structures that are expressed by corresponding algebraic properties of the characteristic semigroup (such as singly generated abelian automata by abelian monoids, and strongly connected state independent automata (see below) by right groups).

A second thread of interest relates to finding less restrictive classes of automata than perfect automata that still exhibit a compelling relation between algebraic and automata structure. One of the ideas that drew significant attention was the group-type automata of Trauth [130] which replaced the abelian condition. His approach also unified and clarified the earlier work of Weeg, Brown, and Barnes [138, 19, 11].

Definition 8 [130]: an automaton A = (S, I, M) is **state independent** provided that for all s∈S, $x \sim_s y$ if and only if $x \equiv y$. That is, each agreement class $[x]_s$ of Weeg is exactly one of the equivalence classes of the characteristic semigroup.

Definition 9 [130]: an automaton A is called **group-type** if A is state independent and the characteristic semigroup S(A) is a group; if A is group-type and strongly connected, it is called **quasi-perfect**.

Trauth provides results showing that the quasi-perfect automata are a useful generalization of perfect automata. In several ways this is an optimal choice to replace the abelian property. For one, A is perfect if and only if A is quasi-perfect and G(A) is an abelian group. A group-type automaton need not be strongly connected. But if it is singly generated, then in fact it is strongly connected (i.e., quasi-perfect). Also, if A is strongly connected and #G(A) = #S, then A is quasi-perfect.

Moreover, quasi-perfect automata have other similar properties to the perfect case. For instance, for a quasi-perfect A, S(A) is isomorphic to G(A). Also, while not all state functions M_x (for $x \in I$) are automorphisms (as in the perfect case), for quasi-perfect automata A, x is in the *center* (i.e., M(s, xy) = M(s, yx) for all $s \in S$ and $y \in I$) if and only if $M_x \in G(A)$. An analogy for the result on direct products of perfect automata was also obtained but was not as persuasive. Trauth's ideas led to an extended variety of other investigations (e.g., [84, 71]).

Another thread in the research was identifying less restrictive alternatives for the strongly connected property that was so frequently assumed for early automorphism results. The most useful and widely studied such property was the idea of singly generated (or cyclic) automata initiated by Oehmke [115]. Oehmke's paper was largely concerned with right-congruences on automata (a topic we do not explore here), but his analysis does lead to the conclusion that if A is singly generated and state independent (and hence strongly connected), then G(A) is isomorphic to a subgroup of S(A).

We have earlier noted that a number of early results that were stated for strongly connected automata stand essentially without change for singly generated automata, and that for state independent automata, the singly generated and strongly connected automata are synonymous.

Bavel not only introduced the alternative term singly generated [12] instead of cyclic, but developed an insightful analysis of homomorphisms using automata structure. For singly generated automata he generalized Weeg's result for strongly connected automata that #G(A) is an integral divisor of $\#S_A$ by showing that #G(A) is an integral divisor of the number of generator states. In fact, when I is a free semigroup, he even improves on that [13].

In a succinct unification, Arbib [3] redeveloped many known automorphism results for the case of singly generated automata, following a variant of the ideas of Oehmke [115]. Recently Tian, Zhao and Shao [128] provided an extensive analysis of the structure and representation of singly generated automata.

The last research thread that we examine in this section concerns the direct product structure of automata. In addition to the early results on direct product of perfect automata, direct product decomposition of strongly connected automata with #G(A) = #S (i.e., the group-type idea that was subsequently named by Trauth) was shown in [40] to be synonymous with a group product decomposition of G(A). Also, Weeg [141] gave a sufficient condition for the direct product of strongly connected automata that $G(A_1 \times A_2)$ is isomorphic to $G(A_1) \times G(A_2)$. More generally, $G(A_1) \times G(A_2)$ is a subgroup of $G(A_1 \times A_2)$.

A uniformly structured decomposition of automata was developed by refining the methods of Hartmanis and Stearns [74], first by Jump [89] using automorphisms and then by Tiuryn [129] using endomorphisms. Masunaga, Noguchi, and Oizumi [103] also provide several significant results on direct product decomposition using both G(A) and S(A), including also noticing that the decomposition of strongly connected, state independent automata is characterized by decomposition of G(A).

The strongly connected property was problematic for direct product analysis. It's immediate to see that if a direct product automaton is strongly connected, then the two factor automata must also be strongly connected. However, the seemingly natural converse is not true. In fact, [41] showed that if the two factor automata are even "similar" in the sense of one being the image of the other by an operation preserving function, then their direct product *cannot* be strongly connected. So in addition to each being strongly connected, a relationship called "strongly related" is required of two factors for their direct product to be strongly connected, and [41] also provided a necessary and sufficient condition for this.

Final Remarks on Automorphisms

The automaton automorphism concept is a means to formalize and quantify symmetries that occur in this model. It provides an analysis orientation that fosters insights into connectivity properties. This chapter provides only limited coverage of the entire body of work that followed from the earliest automata automorphism research. Presentation of the full scope of that work exceeds the objective of this historical account.

While a couple of hundred papers (and several books) related to the topic have been published, interest in automata automorphisms became widely scattered by the mid-1980s. My own attention was already shifting in another direction by the mid-1960s, and my alternative pursuits are described in detail in the next chapter. Although significant interest in classic automata theory (including automorphisms) has subsided, use of automorphisms is re-emerging in a variety of computer science topic areas such as: fuzzy automata [122], cellular automata [23], application of formal methods [149], database [57], error correcting codes [62], and theory of computation [15]. Investigations utilizing automorphisms in their analysis have established their value with unique and valuable insights, and continue to foster contributions in a variety of contexts.

Although my research efforts shifted away from this area quite soon, I continued an interest in automata and formal languages throughout my career. I later published a couple of papers on formal languages and hybrid grammars [43, 45], one on semigroups and automata [44], and another on regular languages [55]. Also, several Ph.D. students whom I supervised wrote dissertations on topics in automata and formal languages. Finally, I frequently taught courses in this topic area, and eventually transformed a large collection of class notes into a published text [53].

Chapter III
Programming Languages Thru the Years[2]

Linguistic Structure for Computer Programs

By the mid-1950s there were a number of efforts at creating programming languages that facilitate the development of computer programs. Several came to my attention through my programming responsibilities at the MSU Computer Lab, and FORTRAN was the most successful. FORTRAN originated in 1954 with an IBM team led by John Backus. A compiler first became available for the IBM 704 in 1957. FORTRAN adopted the level of abstraction of familiar mathematical expressions rather than machine instructions. It relieved the programmer of so much tedious detail, and so greatly increased programmer productivity that its use in many applications was compelling. This advantage soon forced other computer companies to develop Fortran software for their own computers. However, compiler construction was ad hoc and not well understood, and programmers with any related experience were in extremely short supply. So FORTRAN did not become broadly adopted or available on other manufacturer's machines until into the early 1960s. In my case I had occasion to read about the language, and to examine FORTRAN programs, but active use was not immediately possible. Since the MSU Computer Lab operated CDC computers, it was the early 1960s before I had access to an implementation. During the intervening period the language had evolved through three versions (see Backus [6]).

Issues about how to provide machine independent descriptions of programming languages attracted my interest right from the beginning. FORTRAN was initially described in a careful but informal way [82], typically in the context of examples. For instance, a list of six "Formal Rules for Forming Expressions" was given, and in Figure 4 you see rule number 6 from that list.

[2] An earlier version of this chapter appeared in *Turing Tales* by E. G. Daylight, Lonely Scholar bvba, 2016.

6. If E and F are expressions, and F is not floating point unless E is too, and the first character of F is not + or -, and neither E nor F is of the form A**B, then

E**F

is an expression of the same mode as E. Thus A**(B**C) is an expression, but I**(B**C) and A**B**C are not. The symbol ** denotes exponentiation; i.e. A**B means A^B.

Figure 4: FORTRAN syntax rule for exponentiation

One consequence of this informal description was that the most convincing argument one could make for or against the legitimacy of a FORTRAN program containing an uncertain construct was whether or not it worked in “the” compiler. But whose compiler – numerous computer manufacturers were providing a “FORTRAN” compiler, and they varied significantly in what was admissible, and in how an admissible program worked. Although FORTRAN’s position was dominant in those days, this problem seriously troubled its use as it spread from IBM to industry-wide acceptance. This difficulty could make portability of a FORTRAN program between different manufacturers computers as troublesome as writing the program from scratch all over again. This situation persisted until 1966 when the first standard [135] was established (but significant compatibility difficulties still lingered on).

I completed my Ph.D. in 1964, and at this time Dr. Von Tersch was both Director of the Computer Lab and Head of the Electrical and Computer Engineering Department at Michigan State. At this pivotal juncture for me, Dr. Von Tersch offered me a position in the Electrical and Computer Engineering Department with the title of Assistant Professor of Computer Science. This title was a bit remarkable in that a Computer Science department would not officially exist at MSU until four years later (and in the College of Engineering), but I did appreciate that gesture. In my first academic appointment I taught a yearlong advanced undergraduate computer programming course of my own design. And the position included a half-time appointment in the Computer Lab as head of systems programming. This enabled me to continue the software development activity that I had enjoyed and valued while experiencing the stimulating activities that are inherent in an academic position.

On a related note of historical interest, Dr. Von Tersch’s roots had been in Iowa, academically at Iowa State University. Later in 1968 Dr. Von Tersch became Dean of the College of Engineering at Michigan State, a position he held for more than twenty years. He was also instrumental in the establishment of the Computer Science Department at MSU. He died in 2010 at the age of 87.

During that first year of teaching at MSU I learned my next two programming languages. One of these languages was ALGOL 60. ALGOL 60 [109] was developed by an international committee with the purpose of creating an international algebraic language “to describe computational processes”, while remaining “as close as possible to standard mathematical notation”. The committee members drew on experience with existing problem-oriented languages as a basis, but they also introduced new ideas with ALGOL 60. The overarching idea of “block structure” was one innovation. In addition to providing the organizational basis for subroutines, it allowed for nesting of program elements with scoping rules that provided for localization of identifiers at any point in a program. For procedures/functions two distinct means of transmitting arguments (by name or value) provided enhanced generality. Also, there were multi-assignment, dynamic for-statements, dynamic arrays with generalized indexing, etc. Finally, there was one other important feature missing in FORTRAN that was provided and fostered in ALGOL 60 – recursion, and this inclusion was instrumental in bringing recursion into the mainstream (how this came to be is an interesting story, see Daylight [30]). Following its success in ALGOL 60, recursion became a feature of nearly every programming language to follow (even, eventually, FORTRAN). The collection of features in ALGOL 60 provided new challenges for compiler writers and fostered significant research into techniques needed to accommodate them (e.g., [118]). The collection of features, their generality, and their interactions introduced various subtleties that led to the need for a clarifying revision of the defining report [110] three years later, and this is a point that will arise again later.

A crucial part of the process of describing this new programming language was to incorporate a notation to precisely detail what was admissible in the language, and do so in a manner completely independent of any specific compiler or computer. The conceptual idea for the notation was due to John Backus [5], although almost simultaneously there were several closely similar but apparently independent developments that I'll return to later. This notation came to be known as BNF, an acronym for Backus Normal Form, or sometimes alternatively Backus-Naur Form to credit the editor of the ALGOL 60 Report, Peter Naur. Although devised by John Backus, it was Peter Naur who championed its use [111] in the defining report and coalesced the description of ALGOL 60 around BNF. As one small indication of the change that the use of BNF brought to the description of syntax, consider the clarity of the sole compact ALGOL 60 BNF rule (from the revised report [110] actually) to express exponentiation in an expression (using ↑ for the operation): <factor> ::= <primary> | <factor>↑<primary>. And compare that with the earlier example of FORTRAN's syntax rule (Figure 3). The use of BNF for the language description made it not only eminently more clear and succinct but at the same time provided greater generality. John Backus received the ACM Turing award in 1977 for his work on FORTRAN and BNF, and Peter Naur received the ACM Turing award in 2005 for his work on ALGOL 60.

ALGOL 60 was an influential programming language for several reasons. It incorporated programming language features that stimulated years of study and discussion (e.g., [92, 18, 102, 46]) that promoted advances in the area. Through its reliance on BNF, another aspect of ALGOL 60 was the utter clarity of its syntax and its avoidance of quirks and special cases. By this one example, BNF was instantly established as the means to describe programming language syntax. BNF is classified as a metalanguage – a language used to describe other languages. So while it does not go into a list of *programming* languages, BNF is a linguistic construct, and being confident with its use proved to be indispensable to subsequent learning of other programming languages. BNF has been used to describe almost every language developed after ALGOL 60, and it provides a vital tool for pursuing foundational work on programming languages. While the significance of ALGOL 60 for programming languages is profound, experience has shown that BNF has had an even greater and more long lasting impact. It is striking that at this early stage, for the second time John Backus played a critical role in programming language development.

The second programming language I learned during that first year of teaching at MSU was IPL-V. Early versions of the language were developed at the RAND Corporation in the late 1950s. Allen Newell and a group at RAND and Carnegie Institute of Technology developed the IPL-V version [112] (the first public version) that was initially released in 1960 for several IBM computers and subsequently implemented on other manufacturer's machines (including CDC). While FORTRAN and ALGOL 60 were designed primarily with numeric processing in mind, IPL-V was designed for symbolic (non-numeric) processing. It was intended for adaptive problem-solving tasks involving symbol manipulation such as formal theorem proving. The heart of IPL-V was the linked-list data structure – this was a data abstraction that encouraged thinking at a higher level, allowed great flexibility and generality of data organization, and provided for dynamic space utilization. A prominent feature of IPL-V was the inclusion of an extensive library of list manipulation operations that facilitated symbolic computations. Linguistic characteristics of IPL-V were primitive, and I did not use it beyond the teaching context that first year. However, work on IPL-V inspired much additional interest and its operations were to be encountered many times over in subsequent languages (see e.g., [16]). During this time period, linked-list processing techniques developed rapidly by a variety of sources, both practical and formal (e.g., [104]). Years later (1975) Allen Newell was co-recipient of the ACM Turing award, in part for work on list processing.

While there would be a germination period, my early programming experience and particularly my programming languages exposure during my first year as a faculty member, would exert a growing influence on my research activities. My initial research interest in automata theory would evolve over time toward formalisms that were to become vital foundations for programming and programming languages.

Settling into an Academic Career

In 1965 I accepted an offer to join the Computer Science Department at the University of Iowa in its inaugural year. This was at the invitation of Professor Weeg. Professor Weeg had left MSU before I finished my Ph.D., and joined the University of Iowa as the Director of the Computer Center with a joint appointment as a Professor of Mathematics. In a remarkably short time he accomplished the formidable task of gaining approval for a new department in a discipline that was not yet well established! He served as the first Chair of this department as well as the Director of the Computer Center. The newly established department was located in the College of Liberal Arts and had a close connection with the Mathematics Department. My experience in the Computer Lab at MSU led me to attach substantial importance to continued participation in those associated aspects of professional computing activity, and my new position included a joint appointment in the Computer Center at the University of Iowa. Finally, my new position also initially included a joint appointment in the Mathematics Department, a feature that strengthened the alignment with my academic identity at that point. This was a perfect match with my background and interests, and was an opportunity that enabled me to work next to Professor Weeg on a daily basis in a multifaceted environment.

While a variety of courses and programs could be found at numerous institutions, for department-level academic units with ‘Computer Science’ in their name, a list by Ralph London [101] shows only 10 departments existing in the U.S. by 1965, with the first starting in just 1962 (at Purdue University). Ralph London also reports that the first Computer Science Ph.D. at the University of Iowa was Mortesa Rahimi (in 1968), and I note in passing that I was his thesis supervisor.

As the Computer Science Department at the University of Iowa was formed, and throughout its early years, there was extensive nation-wide debate over just what the definition of this newly emerging academic discipline should be. The appropriate curriculum was vigorously debated, and new work that profoundly advanced the discipline was being published constantly. New developments had unavoidably immediate impact on the curriculum at all levels. Substantial course revisions were needed at virtually every offering of most courses, and the introduction of brand new courses continued at a brisk pace. Invariably in this situation, textbooks lagged behind the current state of knowledge, requiring supplementation that further complicated class preparation. While there had been several earlier curriculum proposals, in 1968 the first comprehensive proposal to achieve significant acceptance appeared [4] and this subsequently helped to bring a modicum of uniformity to academic endeavors (though it certainly did not end the debates).

At the time the Computer Science Department at the University of Iowa formed, it had no computing equipment of its own. Both faculty research and programming classes were served by the facilities at the Computing Center, a campus-wide academic service unit. The Center provided a traditional batch-oriented, large mainframe service common during this period. Our center was an IBM shop, running an IBM 7040 at the outset, and acquired an increasingly powerful succession of models of the IBM 360 line beginning in 1965. Students and faculty went to the Computing Center to keypunch their programs and data, and submit their card deck to an operator. The submitted card decks were "batched" and run by Center staff in succession. Sometime later one would return (four hour turn-around was typical for "normal" jobs) to retrieve the deck and a printout. Because of the high system overhead in individual job initiation, large classes necessitated the acquisition of "in-core" translators (e.g., WATFOR [124]) that would internally process a series of small student programs as a single job rather than treating each one as a separate job run. This provided a huge reduction in system overhead, and together with the advance to a multi-processing operating system, led to turn-around of an hour or less for short student jobs. Having a department chair that was also the Director of the Computing Center proved to be advantageous in the competition for software acquisition and access to the single campus-wide computer facility.

The Computer Center at the University of Iowa proved to be a beehive of activity. My formal position there was something like 'head of programming'. But I actually had little opportunity for that activity in the early years. The responsibility as the campus-wide computing facility implied a diversity of needs for service and included some particularly large demands. Beginning in 1958, Dr. James Van Allen in Physics had radiation sensors placed on satellites that produced vast amounts of data requiring analysis. This resulted in the discovery of the "Van Allen radiation belt" and led to NASA contracts to providing funding for the pressing need for computer analysis. Also, in the Education College Everett Lindquist had devised the ACT and other standardized educational tests. To support the broad adoption of these tests Lindquist led the development of the first optical mark scanner to supersede the electrical scanners then in use. Both the tests and the scanners were very successful and created another large demand for processing capacity. Also, the Computer Center was authorized to sell computer time to off-campus entities, both educational institutions and industry. Area industry was quite active and included the Army's Rock Island Arsenal as a major user (anti-Vietnam protests and the Defense Department source of funds eventually brought an end this activity in the early 1970s). And with the aid of NSF grants for high-speed communication links secured by Dr. Weeg, the Computer Center became an active regional center for more than a dozen area educational institutions.

From these diverse obligations for computing services, plus rapidly increasing enrollments in programming courses and expanding use of computing in other disciplines, the need for computing resources continually grew at a pace that threatened to overrun capacity (sometimes doubling in the space of a year). Each expansion would require the purchase of a multi-million dollar mainframe computer more expensive than the last. The process for each such upgrade required creation of a request for proposals based on a thorough analysis of the evolving needs to be met, followed by evaluation of proposals from various computer vendors for their hardware and software configurations, sometimes including trips to their facility. Dr. Weeg had me work closely with him on these efforts and the political pressures to resolve them were often far more pressing than academic matters. Despite a variety of funding sources, expenses of this magnitude were a major hurdle for a state university and required a high priority with the active involvement of the central administration to accomplish. The Vice-President for Educational Development and Research was D. C. Spriestersbach and he bore primary responsibility for university oversight of the Computer Center. Dr. Weeg had a good working relationship with Spriestersbach who enthusiastically accepted the argument that support for research and educational development entailed placing a high priority on making excellent computing resources available, and Spriestersbach provided consistently positive support in the vigorous debate over funding allocations to accomplish timely upgrades.

As a related note of historical interest, Professor Weeg unfortunately developed a brain tumor and died in 1977 at the age of 49. In 1978 the University named the Computer Center in his honor. Spriestersbach devotes a chapter in his expansive recollections [125] that expresses a central administration perspective on the development of computing at the University of Iowa and gives total credit to Dr. Weeg for his insight, inspiration, planning and leadership. Also Dr. Weeg was recognized for his many wide-ranging accomplishments by inclusion in J. A. N. Lee’s list of “computer pioneers” for the IEEE Computer Society [97].

Programming Language Proliferation

FORTRAN, ALGOL and assembly language, plus COBOL, dominated the programming landscape in the U.S. throughout the 1960s. At the University of Iowa the FORTRAN language was used in programming classes starting in the early 1960s, and it continued to be used in the first programming course until 1976. However, by the mid-1960s a number of other languages had begun to appear – e.g., APL, Basic, SNOBOL, and several variations of ALGOL. In fact, a widely referenced "Tower of Babel" cover on the Communications of the ACM in January 1961 listed over 70 languages in common use, and emphatically reflected a rising concern over the language proliferation phenomenon.

In the mid-1960s in response to this programming language proliferation, IBM mounted an effort to unify the language diversity that continued to expand by developing a broad-spectrum language that incorporated features from FORTRAN, ALGOL, COBOL and other languages. The goal was to provide a single language suitable for both scientific and business applications as well as system programming. A team composed of IBM staff members plus members of the IBM user group SHARE carried out the initial design of the language, and it was initially called NPL [117]. Due to a conflict with a prior use of this name, this language was soon renamed as PL/I.

PL/I freely adapted features from the languages indicated above and integrated them in a coherent way. But it was innovative in expanding with features not available in its predecessors. It provided three classes of storage allocation – static, automatic (stack-based), and programmer controlled (heap-based). It also included exception handling features and rudimentary multitasking. It included string handling, pointers and complex numbers. Extensive defaulting conventions were included and all of its large set of keywords were also available as program identifiers. These features greatly complicated the task for compiler writers. In his Turing Award presentation [78] Hoare, who participated in the PL/I design process, recalls his dissatisfaction with the ambition of both the design process and its results. He speaks of unsuccessfully urging the elimination of features viewed as “dangerous” and expresses the judgment that PL/I was a “technically unsound project”. The PL/I project represented an inflection point in programming language design and evoked the misgivings of some.

IBM developed an implementation of PL/I (released in 1966) for its machines and the language was quite successful in that context. As the language’s popularity spread, it began to be adopted for teaching by a growing number of American universities. However, in those days both the typical mode of computing (“batch processing”) and the size and poor efficiency of the IBM PL/I compiler made it impractical for large classes. This led Cornell University to develop (on IBM computers) an efficient in-core compiler system PL/C [108] suited for this purpose. In 1970 the beginning programming course at the University of Iowa was expanded into a two course sequence. The first course continued to be taught with FORTRAN, but with the support of the PL/C system, the second course in the sequence used PL/I. Later in 1976 the first course of our beginning sequence also began using PL/I.

In the fall of 1969 a Conversational Programming System (CPS [83] – an IBM “type III” unsupported program) became available. Despite its status, CPS provided an ambitious, high-quality interactive system implementing the PL/I language. As an interpreter, CPS offered incremental compilation and accepted program fragments with immediate execution feedback. Unfortunately, resources (e.g., terminals and processing capacity) were only sufficient to allow Computer Science faculty early access to this system. It was immediately obvious that this interactive paradigm would provide an enormous advantage for program development. Interactive access for general faculty access and class instruction would come significantly later.

Development of implementations of PL/I by other computer manufacturers was problematic, and by 1980 interest in the language had declined substantially. While PL/I did not achieve its motivating purpose (language proliferation subsequently accelerated rather than declining), it had a significant impact on programming language ideas, goals and ambitions. The successful implementation of this language encouraged subsequent language designers to elevate linguistic matters and reduce the emphasis on efficiency when considering language goals. This influence has led to noteworthy contributions to the development of programming languages, of course, all made feasible by amazing increases in machine capacity.

In the latter half of the 1960s another language attracted a good deal of my attention. This was the language EULER, introduced in two papers published by Niklaus Wirth and Helmut Weber in 1966 [144]. This language was a derivative of ALGOL 60 and provided generalizations of some features, although it abandoned static typing. Its primary attraction to me was the inclusion of a complete formal (and machine independent) definition of *both* the syntax and semantics. BNF provided the standard means of accurately describing programming language syntax, but was not directly useful to describe semantics. Wirth & Weber developed a formal means of precisely describing semantics and used EULER as their case study demonstration. They attached the generation of code for a virtual, list-oriented machine to the parsing process for a restricted type of grammar. Each production of the grammar for EULER was attached to a code fragment that described an execution effect. As the parsing process identified the occurrence of a production, it also generated the corresponding code. This research was an elaborate harbinger of Knuth's attribute grammars [93] although it avoided the construction of a derivation tree by direct augmentation of the parse procedure, and thereby could only operate with what are known as "synthesized attributes" in attribute grammars. However, this was sufficient for a comprehensive semantic description, and demonstrated how to avoid later clarifications like that necessary for ALGOL 60.

For several years, I used EULER and its associated formal description in a graduate course on programming language foundations. It provided a superb illustration of the complete formal description of a language. The language description included detailed operational development for parsing, code generation, virtual machine and interpreter, providing a model suitable for creating an implementation of EULER. This was done under my oversight at the University of Iowa (and by others at other universities), allowing experimentation with some relatively intricate EULER programs. The EULER description exposed the complication and subtlety that derive from ALGOL's block structure scope rules, call-by-name parameter transmission, procedures as arguments, etc. I gained much better insight into these issues by augmenting the study and use of the EULER description with that implementation effort, and I believe it was a fruitful educational tool. While there were a number of efforts at connecting formal specification of language syntax and semantics with implementation, I regarded the EULER project as a model of conciseness and completeness.

Although in the early 1970s, the first programming course at the University of Iowa still taught FORTRAN, a new wave of language change was on the horizon. Two programming languages appeared that were destined to have a profound effect on computer programming. These languages were C and Pascal. However, I did not seriously pursue either of these languages initially and so I'll defer any comments relating to them until a bit later. Instead, I became interested in a language that embodied a distinctly different paradigm – SNOBOL – and this is the direction I pursue next.

Advancing the Theory of Programming Languages

As briefly alluded to earlier, in the early 1960s a variety of formalisms for language description began to appear that would have profound effects on the design and implementation of programming languages and systems. The regular expression concept from automata theory had already undergone substantial theoretical investigation, and had obvious potential for practical application. The use of BNF to describe ALGOL 60 instantly established it as a mainstay for programming language syntax. The Chomsky hierarchy of grammars and languages devised for the analysis of natural languages had immediate relevance to programming languages. And lastly, the development of the programming language SNOBOL provided a cogent practical alternative for describing collections of strings.

Regular expressions first made their way into text editors and operating systems [127] and evolved into a substantially more expressive mechanism than the theoretical expressions that Kleene [91] had devised. This mechanism together with common extensions has no universally accepted name, but is often referred to as *extended regular expressions* and that is the term I will use in this document. Consequently, the well-developed theory of regular expressions from automata research has limited applicability to the extended regular expressions that have come to be part of numerous programming systems. These enhanced mechanisms are a key component in a long list of modern programming languages, editors and systems [60].

Chomsky's work on grammars [24] provided a rigorous framework for formal language research and this blossomed into an active research area. I followed the theoretical side of the research on formal grammars diligently as it progressed. One of the grammars in the Chomsky hierarchy, the context-free (or type 2) grammar turned out to be a direct match with BNF, and its formal basis applied immediately to languages described by BNF. As a result, a variety of theoretical properties, plus ideas of direct practical use such as syntax trees and parsing methods were soon being adapted to programming languages (see e.g. [58]).

At the heart of the SNOBOL language [35] was a construct known as a "pattern" that manifested a distinct alternative to describing and processing strings. The SNOBOL pattern concept anticipated the enhancements to regular expressions that would make them so practically useful, and included several more features of significant practical value. The aim of SNOBOL was practical rather than theoretical, and practical programming ideas preceded any theory. But it was a relationship with other theoretical investigations I've noted that drew my interest, and I'll discuss these connections next.

SNOBOL was begun in 1962 by a group at Bell Telephone Laboratories led by David Farber. It went through a sequence of versions [70] before a widely available implementation of SNOBOL4 [68] became available for the IBM 360 System in 1968 when it attracted my serious attention. SNOBOL4 was intended to facilitate string processing, and provided a drastically different programming paradigm. The central feature of the language was the string *pattern* and the closely associated pattern matching process. By providing access, processing, and reuse of substrings encountered during the matching process, SNOBOL enabled succinct and flexible string processing in a way not found in any other language. Moreover, SNOBOL provided a clean and intuitive syntax, and embedded it in a success-failure control structure with automatic backtracking that was very natural to this processing paradigm. The emphasis on string processing did not lead to the exclusion of basic features of a general-purpose language. SNOBOL4 included integer and floating point arithmetic, arrays, tables, and programmer defined data types. By 1970 I was using SNOBOL4 in an upper-level undergraduate course on concepts in programming languages.

The semantics of SNOBOL4 patterns were intimately connected to the matching process, and the success or failure of matching guided the flow of control with automatic backtracking in the matching process. The matching process was described in an entirely operational way [69] by an elaborate description of the scanning process and how each pattern element moved a cursor through a subject string. This included pattern elements that stored a partially matched substring in a program variable for later use (in the same pattern). Nonetheless, this paradigm had the potential for a much more declarative view. A basic aspect of a SNOBOL4 pattern was that it constituted the description of a collection of strings. It seemed of eminent value to me to gain insight into the limits of the kind of collection that might be describable through various types of pattern elements, and this motivated me to develop a formalism suitable for theoretical study of the pattern concept.

In 1971 I published a paper [42] that began with a formalization of the linked-list concept familiar from IPL-V. Using this formalism I showed that one natural view of the sequencing of the elements appearing in a list structure was captured by the language of a context-free grammar whose structure is effectively isomorphic to that of the list structure. This formalism was then extended to capture SNOBOL4 patterns that include embedded defined functions and their application to matched (sub)strings. With this extended model, and using only finite-state string mappings for embedded functions, another result showed that any recursively enumerable language could be described. I found such theoretical insights to be of considerable practical value in programming problems involving pattern design, both through better general understanding and in the identification of pattern elements to be employed. This line of investigation of declarative semantics of patterns became a long-range research topic for me and resulted in several more papers in subsequent years [47, 100, 49]. In passing, I note that not until recent years has analogous theoretical investigation of extended regular expressions been seriously pursued (e.g., [22, 1, 59]).

Evolution of Types and Control Structures

Niklaus Wirth had joined the IFIP Working Group to define a successor to ALGOL in 1964. He eventually submitted one of two proposals to the group, but the competing proposal was selected. Wirth's proposal had included Hoare's suggestion [75] for dynamic records as an ALGOL extension. After the rejection by the IFIP Working Group, Wirth and Hoare independently pursued this proposal and produced a language that became known as ALGOL W [145]. This language was only implemented on IBM 360/370 computers, but I did briefly use it in our advanced undergraduate concepts course. ALGOL W was interesting for its development of types. The original FORTRAN [82] did not have types – it distinguished only fixed point and floating point values and variables. ALGOL 60 did incorporate a type concept, but was nearly as limited with only 'integer', 'real', and 'Boolean' declaration of variables. The ALGOL W proposal expanded the basic types of ALGOL 60 with 'complex', 'bit', 'string', and 'reference' types, and added a dynamic structured record type intended to accommodate list-structure programming provided earlier by languages such as

IPL-V. ALGOL W therefore provided a significant advance in giving type concepts a central role in programming languages, providing a preview for what we next see in Pascal.

By the latter half of the 1960s a movement to “structured programming” was beginning to gain broad support (e.g., [17, 31, 32]). Concern over unrestrained structural organization of programs [31] led to the use of single-entry/single-exit control features as a primary ingredient of this methodology. Pascal was the creation of Niklaus Wirth [146] and was first announced in a 1970 technical report. The Pascal language provided strong support for structured programming with a simplified ALGOL 60 for-statement, and added while-statements, repeat-statements, and case-statements. Also Pascal expanded the steps taken by ALGOL W on the role of types. The reference type (changed to ‘pointer’) was given broad applicability, ordinal and ‘char’ types were included as simple types, enumerated and subrange types were added as was a new set type. And Pascal added a means for programmer-defined types. Pascal required type declarations for all program entities, and continued the strong static typing philosophy in compilation to enhance efficiency and prevent an operation or procedure being inadvertently applied to unintended data.

The diverse combination of control and data structures made Pascal an attractive language for a wide variety of applications. None-the-less it was rather slow to attract adoption in the U.S. In the U.S., one factor was overcoming a large base of established software available for commercial languages. Another factor was that the initial compiler created by Wirth was for CDC computers that were not that common in U.S. industry. The most significant influence for the success of Pascal was the availability of a free and easily portable implementation known as the P-code compiler [113]. Following the strategy used with EULER, a virtual computer was devised with an associated Pascal compiler so that all that was required for a Pascal implementation on another computer was writing a P-code interpreter. While this led to slow execution times that impaired industrial adoptions, it did not significantly deter an academic enterprise, and a free implementation plus the language's attractive features were irresistible in that arena.

As its popularity spread and implementations proliferated, the call for a standard for Pascal gained much support. One central feature of ALGOL 60 and its descendants, including Pascal, is the idea of block structure and its use as the natural scoping unit. A colleague and I had noticed that as the variety of kinds of elements in Pascal had grown and interactions abounded, not all the language rules honored the scoping role of block structure. We believed it would be easy to implement a resolution of the offending cases we had observed, and made a contribution to the standards discussion [9, 10].

The totality of types in Pascal had become quite diverse with enumeration and subrange types, arrays that may be 'packed' or not, variant and normal records, a new set type, plus an added means for the programmer to define types. Throughout its literature (e.g., the Pascal "bible" [86]), the phrase "same type" is used with no elaboration when expressing strong typing requirements, and while the standard [2] gave greater care in describing "type compatibility", the "same type" phrase was carried forward in numerous places. The purpose of strong static typing serves efficiency of implementation. But just as well it avoids errors of applying inappropriate operations to data values as early as possible, and this is an essential characteristic of Pascal typing. I remained with concern that "correct" typing so at the heart of the language might be left with possible room for interpretation. For instance, with the declarations shown in Figure 5, could it be that variables A an B have the "same type" while variables C and D do not?

```
var A: array[1..2] of integer;
var B: array[1..2] of integer;
var C: record x,y: integer end;
var D: record x,y: integer end;
```

Figure 5: Pascal declarations

Later, I pursued an alternative to type equivalence in Pascal. In mathematics when two things are equivalent (i.e., “the same”) there are basic expectations for this relationship and it seems those properties should apply for Pascal types as well. Thinking about type equivalence led me to an idea for a somewhat more flexible approach that would still be straightforward to implement. My approach correlated with a natural “subtype” idea, and I developed precise definitions and an associated type checking algorithm in [50].

Two important design goals for Pascal [147] had been to serve as a sound language for teaching, including teaching system programming, and to achieve a high level of efficiency for both the compiler and compiled programs. As the success and longevity of the language clearly establishes, these goals were not only attained but exceeded. In 1982 Pascal became the language used in the beginning two-course sequence in programming at the University of Iowa, and it remained in that role for more than a decade! And Pascal was the programming language of choice at the beginning of Knuth’s campaign for literate programming [94]. In 1984 Niklaus Wirth received the ACM Turing Award for his contributions to programming language development.

Distinctive Language Alternatives

Languages that adopt an unorthodox perspective on computing have often held an attraction for me. Of course, just getting off the beaten path isn’t necessarily an accomplishment. But internalizing a truly different approach does open avenues to new thinking. When facing an unfamiliar programming task, having diverse options enhances finding good choices. SNOBOL is one such language that I have already discussed. In this section I discuss several others that have expanded my computing perspective.

Kenneth Iverson developed a “programming notation” he called APL while teaching in a graduate automatic data processing program at Harvard in the latter half of the 1950s. His notation was heavily based on mathematics and matrices, intended as “a tool for communication and exposition” of algorithms [34], and motivated by dissatisfaction with then current programming notations. In its early years, APL was used as a means to communicate between individuals rather than as a *computer* programming language and this was immediately apparent in its appearance. Although I did not pursue APL until much later, the first widely available documentation was published as the book “A Programming Language” in 1962 [85]. As a brief illustrative example that distinguishes it from a programming language, one statement appearing in the APL book (p. 44) is shown in Figure 6.

$$\boxed{{}_{u_2}\Phi^{\bar{u}_2}_{u_1} \leftarrow {}_{u_2}\Phi^{u_2}_{u_0}}$$

Figure 6: an APL assignment statement

The APL language has at its heart the array data structure with no restrictions on dimensionality. It has operations to conveniently construct arrays, to adjust the number and range of dimensions, and many array-oriented operations. And the basic scalar operations have their definitions extended to apply equally well to arrays of any dimension. For instance, for two linear arrays $x = (x_1, x_2, \ldots , x_k)$ and $y = (y_1, y_2, \ldots , y_k)$, the expression x+y yields the array of element-wise sums. In a conventional language one would need to introduce an index variable and write a loop with a suitable termination test to describe the same computation. APL programs through this and numerous other conventions are both remarkably more succinct yet still improve clarity. Iverson describes the name APL as an acronym for “A Programming Language”, but when I think about its profusion of array facilities, it’s Array Programming Language.

Iverson joined IBM in 1960 where he initially collaborated on using his language as a means to describe computer hardware. It was not until the mid-1960s that serious effort began on developing a computer implementation of APL. This was challenging for several reasons. The first problem was that APL used a variety of math symbols, Greek symbols, special marks, and 2-dimensional spacing. Resolving this involved the development of a new terminal based on the IBM Selectric typewriter, plus a specially designed APL type-ball that provided a tailored 88 character alphabet with many of the unique APL characters (but only upper-case letters). Even so the implementation would still require an overstrike procedure for some operations. And with all that, the “linearization” of the notation required language adjustments. But despite these changes, the implementation preserved the original sense of the language quite well. In 1968 IBM released a version of APL for its System 360 series of computers [33], and it was with this interactive system that I become familiar with APL.

The often mentioned “one-liner” phenomenon of APL is a topic that gets us directly to the quintessence of the language. This is an informally defined term that refers to the composition of a selection from the profusion of APL operators to form an expression that accomplishes a non-trivial computation. I’ll illustrate what’s meant with a single but informative example – computing the prime numbers less than a given bound. This is a common exercise in many languages. It is one I first did on the IBM 650 (but not this way), and I was intrigued to see there is a published history of this programming problem [21]. Imagine this program for a moment as written in your favorite language. The point is that this is *one line* of APL code shown in Figure 7. I present it here for an insight into APL, and as another instance of the distinct appearance of APL programs.

```
(~T∊T∘.×T)/T←1↓⍳N
```

Figure 7: APL primes program

This is not the place for a detailed account of APL operators and syntax, but a short informal description of the primes program in Figure 7 does reveal the alternative view that APL fosters. Given integer N, this program (read right-to-left) first creates a vector of values 2 through N, from that it creates the 2-dimensional matrix of all products from this vector, then it next selects occurrences in the original vector of integers that appear in this matrix (these must therefore be those values which are products), and finally inverts that selection to obtain the list of primes! So, no (explicit) loops, tests, etc.; no quotients and tests of remainders, just the creation of arrays, multiplications and selections (each by a single operation), and the correctness is self-evident! This program is a precise computational description of the primes and conveys the sense of APL's earlier use for communication between individuals. Of course, this program clearly overlooks consideration of both processor and storage efficiency, a common criticism of APL.

In 1979, Iverson received the ACM Turing award for his genuinely innovative ideas in programming. But he had a noteworthy encore that deserves greater recognition than it has received. This requires a brief chronological fast forward, but it best fits here. In 1989, Iverson collaborated with Roger Hui in the development of a language for an ASCII-based descendent of APL known as J [80]. The special characters for operators were replaced by a systematic and well-conceived pairing of ASCII characters, and intrinsic APL facilities were carried forward into J. But in addition to selecting a syntactic translation for APL, the language evolved in significant ways. The APL idea of array shape was generalized in J, and the idea of array rank was extended to functions, providing numerous opportunities for both generalizing and simplifying array operations. Although APL was not a functional language, it had a substantial functional core and in J this functional subset is expanded and emphasized. And free implementations of J are available for all the common platforms [88]. I have been fascinated to find an implementation available for my smart phone, where J's continued propensity for "one-liners" is a perfect match for the small screen.

In the mid-1970s the idea of "abstract data types" (ADTs) attracted growing attention. Actually this term was used to describe two related but quite distinct ideas. The first was concerned with programming language facilities to allow users to code "first class" data types that have equal footing with those native to a language (e.g., [98, 87, 148]). This was quickly followed by an alternative idea for the development of methodology for *specification of* the *behavior* of a data type (e.g., [99, 65, 72]). Specification efforts were concerned with program proving [76, 77], and with precise descriptions of new data types suitable for that purpose while remaining devoid of implementation assumptions. The term *algebraic abstract data type* (algebraic ADT) was used for these ideas and this became a central approach for specification, and this area expanded rapidly in the latter part of the 1970s. I found it remarkable that without choosing a representation for data nor providing implementations for the operations, a manifest description of essential properties was still attainable. The algebraic ADT requires only names for relevant type domains, operation names (i.e., for functions with no side-effects) involved along with their type characteristic, and equational axioms for operation outcomes.

To augment this vague algebraic ADT outline, a token version of the standard example of the pushdown stack appears in Figure 8. Note that the specification expresses terms that describe each Stack, and it allows the essential last-in-first-out behavior of operations on a Stack to be deduced from the equations (for brevity we ignore the issue of top(new)).

```
Domains: Stack and Item (not otherwise elaborated)
Operations:
  new: Stack   – result (a constant) is an empty Stack
  push: Stack × Item → Stack – result Item added to top
  pop: Stack → Stack   – result has top item removed
  top: Stack → Item   – result is the top Item
Axioms (for all s∈Stack and i∈Item):
  pop(push(s,i)) = s
  top(push(s,i)) = i
```

Figure 8: Pushdown stack algebraic ADT

As a specification device, algebraic ADTs proved to be remarkably effective for a vast variety of data type descriptions. This is another instance of a device that while not a programming language, has such a close connection that it is appropriate to include in the discussion. The adoption of algebraic ADTs for specifications soon led to another problem. For more complex examples, the creation of the algebraic specification is itself a challenging technical task. While a suitable specification provides a beneficial means to gauge the correctness of code, an erroneous specification can have the doubly harmful effect of leading to incorrect code but believing it correct. Just as we need to debug code by running it, it was soon found that computer aids were needed to discover unintended faults in specifications. Hence several projects to develop systems to automate the verification of an algebraic ADT specification arose (e.g., [114, 73, 66]). These systems provided execution behavior given only the algebraic ADT and further blurred the line with programming languages. While such tailored systems have some clear advantages, I observed that a widely available programming system, SNOBOL4, already provided support directly applicable to this task. I wrote a paper [48] that developed the incorporation of ADT equations as SNOBOL4 code that allowed for testing of a specification, and included suggestions for dealing with some subtle issues that can arise in certain situations.

Through their clustering of a set of operations in direct proximity to their data, abstract data type ideas (both facets) were a factor in the development of object-oriented (O-O) programming. But the first language with objected-oriented features, SIMULA [27], had anticipated this view by the mid-1960s. This language was devised to enhance ALGOL 60 with facilities to aid in the creation of simulation programs. It was quickly recognized that those ideas were of much more general use, and the next step was SIMULA 67 [28] which provided objects, classes and subclasses, and inheritance similar to those found in many O-O languages we see today. And the relation of SIMULA 67 to specification and proving ideas was soon developed by Hoare [77] with a continuation of his earlier program proving work. Although I had followed ALGOL 60 closely, I never did pursue programming in SIMULA. In 2001 Ole-Johan Dahl and Kristen Nygaard received the ACM Turing Award for their original development of object-oriented programming.

My first encounter with O-O programming was Smalltalk-80 [67]. Work on Smalltalk began in the early 1970s and it went through several versions. In his discussion of the history of Smalltalk, Alan Kay [90] relates a wide range of factors influencing its development. Kay mentions the Sketchpad system [126], ALGOL and EULER, and work on porting an implementation of SIMULA. But the other factors he includes range to the broader topics of a changing computing environment with personal handheld computers, high-resolution displays, and windows based systems. There are certainly a wide array of technical and social factors that have made O-O programming so prominent, and fostered the explosive development of new O-O languages that has persisted ever since.

In the Smalltalk-80 description [67] the authors write that “Smalltalk is based on a small number of concepts”, but I have to disagree. They base their contention on a list of five key words in the Smalltalk vocabulary: object, message, class, instance, and method. In fact, each of those words itself embraces a number of concepts. The word “object” involves concepts of data storage, operations upon that data and associated clustering, and some objects involve such ideas as the file system. The word “message” involves concepts of activation, of expression, and communication through concepts of argument and transmission of arguments, etc. The authors themselves essentially concede that the claim is an exaggeration when they write: “These five words are defined in terms of each other, so it is almost as though the reader must know everything before knowing anything”. But as the strategy for organizing all those ideas, the unwavering adherence to the object-oriented methodology reflected in their five words does indeed lead to a remarkably homogenous and coherent result. Another consequence is a simple, easy to learn syntax. The last component is a class library that provides well-conceived organization. Where program libraries in conventional languages are adjuncts of occasional use, the class library is integral to Smalltalk. The class library also encapsulates the primary source of complexity in the system and so allows much of that to be internalized gradually. In 2003 Alan Kay received the ACM Turing Award for his work on the development of Smalltalk.

My experience with Smalltalk-80 was mostly in its use as a teaching system. I began using it in an advanced undergraduate course on programming language concepts in the latter-1980s. The initial Smalltalk-80 implementations were only available for interactive platforms with high-resolution graphics, and these machines were not feasible in our classroom setting at the time. The system I initially used was Little Smalltalk developed by Timothy Budd [20] who in facing a similar circumstance, created an implementation for Unix[3]-based systems that was freely distributed. This system made concessions to run on text terminals, but the essential sense of the language was still evident.

Much later I published a paper [54] that presented a means to enhance algebraic ADTs to provide specifications for object classes. In their usual role, algebraic ADTs model collections of functions – there's no shared state or side-effects. My idea retained the equational character, and based the specification on sequences of messages rather than individual messages. This captured the shared store at the same level of abstraction as functions, and provided modeling for a changing store without a prescribed configuration of instance variables, just behavior of methods. The paper illustrates that a specification given in this way can be used for the verification of varied implementations and storage configurations, using Smalltalk as the programming language.

Declarative Languages

The "declarative" adjective could be applied to numerous languages these days. I won't pursue a careful definition, but for me the key issue is the focus on results, and the key feature is variables as they're known in mathematics not conventional programming languages. The declarative languages emphasize results over process, and have commonalities with program specification such as the algebraic ADT I've already mentioned. In this section I discuss three programming languages I've had occasion to consider previously in a case study comparison [52].

Functional programs in the form of lambda expressions [25] were already there at the dawn of the theoretical foundation of computer science, preceding computer programming. Later LISP [104] was an early version of (impure) functional programming with a version

[3] Unix is a registered trademark of The Open Group

of lambda expressions. While conceptually interesting to me, I found LISP syntactically unattractive and never became motivated to pursue it seriously. As functional programming evolved, it remained a programming niche for me that I never quite made time for. A strong interest in functional programming did not develop for me until reading John Backus' Turing award paper in 1978 [7]. I found his rationale and motivation for functional programming very persuasive. And the algebraic flavor of his FP language resonated with my earlier mathematics background. The direct application of algebraic analysis to reason about and transform these programs was striking. Both the conciseness and the reusability characteristics of functions in this setting were highly attractive. And it was remarkable that his advocacy for functional programming was combined with a strong condemnation of traditional languages for their defects, especially since the award was given for his fundamental work on these earlier languages.

For some time I followed FP related literature. I developed a local implementation of FP suitable for experimentation, and employed it for several years in an advanced undergraduate class on programming concepts. My interest continued to develop as I explored the contrasts in this paradigm. I found the computations that could be expressed using FP's 'insert' operations to accomplish looping (with no explicit iteration or recursion) quite remarkable. Backus and his colleagues had noted this in their writing, but it seemed of even greater significance to me. This was the same operation known as reduction in APL, and the FP insert-based programs were reminiscent of APL one-liners. These FP programs employ conditionals and insert, but avoid explicit loops and recursion. This gives them a "tree structure" with linear execution along each path, and they embody a certain essence of FP programming. Eventually I published a paper [51] that provided a theoretical basis that identifies the computational expressiveness of these "insert-based" programs, showing that every primitive recursive function (e.g., see [29]) could be programmed in this way. This is an extensive class of total functions that in its day was regarded in theoretical literature as including much of practical computing, and whose program counterparts are guaranteed to always halt. My interest in functional programming was significantly heightened the following year (1987) when I attended the Institute of Declarative Programming at the University of Texas (Austin), one of the University of Texas Year of Programming series (see Fig. 9; source: personal collection). I was motivated to hear John Backus speaking about recent work by his group in functional programming [8]. But what actually most caught my attention in his talk were his highly complementary comments about David Turner's work in functional programming.

Fig. 9 : UT Year of Programming participants, Aug. 1987

David Turner had developed a series of three functional languages, plus devising a novel implementation technique for functional programming [132]. I subsequently focused on Turner's most recent language, Miranda[4] [133]. This was a pure functional language but with extensive supporting features. I was impressed by the conceptual foundation for Miranda where a program consists of defined functions and data, applying functions to arguments, and that's it. Both arguments and results of functions can themselves be functions. And the syntax was elegant, providing pattern matching for clean integration of multi-case definitions, "off-sides" rules for flexible but clear multi-line definitions, etc. Miranda also includes the 'insert' operation of FP (or reduction of APL), called 'fold' (two versions actually), providing the means of functional programming that I found so fascinating in FP. Miranda included a strong yet flexible static type system, an attribute I believe contributes significantly to avoiding errors in programs, and one that FP sidestepped. However, Miranda maintains an ease of use by not requiring type declarations but leaving the implementation to perform type inference (another application of the unification algorithm) to determine types. Also included were facilities for both abstract and algebraic data type definitions. I found it intriguing to see ALGOL 60's idea of "call-by-name" parameter transmission carried to logical conclusion in every feature of Miranda via "lazy evaluation". I was further impressed by a simple and practical comment convention that provided an effective means to capture the spirit of literate programming right in the language.

In conclusion for Miranda, I'll (re)consider the prime number program that was presented earlier as an APL example (Figure 7). If we follow the method used in APL, a Miranda version is the function definition shown in Figure 10.

```
primesTo n = [p | p <- [2..n]; ~member prds p]
                where prds = [q*r | q,r <- [2..n]]
```

Figure 10: Miranda primes program

[4] Miranda is a trademark of Research Software Ltd.

Miranda is not as succinct as APL as it lacks such a breadth of pre-defined operators. But this program still replaces explicit loops with a pre-defined list comprehension operation and list membership predicate, and retains the appearance (and clarity) of straight-line code (a “two-liner”). The correctness is as self-evident as for the APL code (more so I’d say), but it’s the same method so the efficiency issues remain. The clarity and coherence of Miranda is remarkable. I regard it as the best language design I have encountered and still enjoy programming with it. I note in passing that a Miranda program was used in the case study paper mentioned at the beginning of this section, and in 1989 the execution of that program took 50 minutes on our departmental (Digital Equipment Corp.) VAX-11/780 and now takes less than one second on my personal iMac.

This brings me to the last language I will discuss. Prolog was initially devised in 1972-73 in a cooperative effort led by Alain Colmerauer and Philippe Roussel [26] with a primary motivation of natural language processing. A more mature version with broader goals began wider distribution in the mid-1970s. However, I remained only vaguely aware of the area for the next decade. In 1965 (John) Alan Robinson had published the celebrated unification algorithm and resolution principle [120]. This was a major advance in automatic theorem proving, and provided a basis for the creation of the Prolog programming language and the initiation of logic programming generally. Alan Robinson became the founding editor of the Journal of Logic Programming in 1984. In 1985 he spent a week visiting in our department and lecturing on logic programming (esp. Prolog). Alan was very generous with his time throughout his visit, and an entirely gracious guest. His presentations were inspiring, and his visit was directly responsible for the development of a long-term interest in logic programming by myself and several colleagues and graduate students.

Prolog’s basis in logic and subsequent selection of computational elements as relations rather than functions was a generalization with a persuasive potential to enhance expressiveness. One program can provide the computations of several programs in conventional (or functional) languages. Prolog's reliance on deduction holds the promise of "smarter" programs. And I found Prolog's backtrack search strategy reminiscent of the SNOBOL pattern match procedure, and it therefore felt like meeting an old friend. So I was quickly won over to this language. Subsequently several faculty and graduate students began a regular seminar on logic programming where we read a wide range of the available literature. Before long logic programming became one of the topics in our advanced undergraduate course on programming language concepts. I recently found it interesting to note a comment by Alan Kay in his history of Smalltalk [90]: "It is a pity that we did not know about PROLOG then or vice-versa; the combinations of the two languages done subsequently are quite intriguing".

Somewhat later one of my Ph.D. students and I developed the use of logic programming as a means to enhance the attribute grammar mechanism for semantic description of programming languages [121]. Rather than taking attributes as data values associated with derivation tree nodes and describing them by functional semantic rules, we advocated using predicates as the attributes. Then the attributes become active rather than passive elements, and the semantic rules are clauses describing the properties of these predicates. This enhances expressiveness with the generality of relational programming, and logic variables provide bidirectional communication that allows tree traversal (synthesized vs. inherited attributes) to be effectively transparent. Moreover, Prolog support for DCG grammars allows syntax rules to be explicitly and coherently embedded within this one formalism.

In the latter part of my teaching career, I undertook the effort to use Prolog (with a strong emphasis on underlying logic) as the language in an "experimental" first programming course for incoming freshmen at the University of Iowa. While I was uncertain at the outset, this course proved to be a popular success with the students who took it. I taught the course for six years, and soon became convinced that this is a superior way for students to develop conceptual foundations prevalent in programming, and of value in a variety of languages that might be later pursued, while becoming familiar with a tool of lasting practical utility. I described the approach I took in [56]. I'll end with one last note in passing. A Prolog program was also used in that case study paper mentioned at the beginning of this section, and in 1989 the execution of that program took 7 minutes on our departmental VAX-11/780 and now takes less than one second on my personal iMac, and a preferred version of that program which would only abort for lack of sufficient space on the VAX-11/780 now completes with no difficulty on my personal iMac.

Summary and Conclusions

As my professional career progressed, my realization that programming languages are much more than just tools to express our preformed ideas continued to grow. As we internalize a programming language, its form and structure molds our thinking at that level of abstraction. Creating a program is a process that requires bridging from a higher-level problem domain to a precise description about how a computation is to unfold within the conception of our adopted language. That intellectual process cannot avoid being shaped by the ideas embodied in the language used for its expression. Of course, we may conceive of a computational approach to solving a problem at an abstraction level apart from any programming language. But when it comes to getting a computer to carry out our approach, we will think of its solution all over again with our internalized knowledge of the organization and facilities provided by a chosen programming language providing the basis at every step. The thoughts underlying both construction and verification of our program can only occur to us by virtue of the structure, organization, and properties of the language that we have assimilated. So it is not just the expression of ideas, but also the very formation of our computational ideas for which we can thank our programming language.

The evolution of programming languages has led to an increasing distance between the programmer and the details of a particular computer. While a universal Turing machine [131] may be sufficient for any computation, it is through more artful languages that we achieve “better” programs. As has been long realized, the metrics for “better” programs may be highly varied and so the proliferation of programming languages will surely continue. This is a phenomenon to be celebrated, as I foresee no limit to the progress that may be achieved. However, it may be challenging to determine which of numerous new ideas are actually “better”. This is where I believe that solid knowledge of what has occurred in the past can improve our judgment.

In conclusion, I summarize those languages whose learning had the most significant impact on my personal understanding and appreciation of programming languages. This leads me to highlight nine of the languages I have mentioned. Each of these languages illustrates that the use of the collection of elements it incorporates leads to a distinctive conception of the programming process. The experience with each of them had a predominant influence on my understanding, and I briefly mention my reasons for each selection. I believe that each of these languages warrants broad general recognition for its contributions to fundamental programming language ideas, and that such general recognition is of value to further progress. However, I express my selection of these nine languages based on the importance of their influence in my personal development:

- FORTRAN: as the most successful early example to raise the level of abstraction far above that of the particular computer to carry out a computation, and encompassing a majority of features found in higher-level languages for years to come
- ALGOL 60 and BNF: for the convincing presentation of a means to precisely describe a systematically coherent, general, and flexible syntax, for nested scoping constructs, and for embracing recursion
- EULER: for demonstrating a methodology for a description of semantics of traditional languages that is both formal and practical, and for demonstrating the practical utility of highly restricted (i.e., precedence) grammars
- APL and its ASCII-based descendant J: for extensive development of an array-oriented language that provides a conceptually distinct paradigm for computing

- SNOBOL4: for an alternative paradigm based on backtracking and success/failure control structures, text processing, and generalized regular expressions
- Smalltalk-80: as an exemplary object-oriented language
- FP: simplicity and power of functional programming, the vast extent of programs that can be written without either iteration or recursion, and the suitability of algebraic methods of reasoning about programs
- Miranda: elegant functional programming, strong static polymorphic typing with type inference, and lazy evaluation used to great advantage in enhancing expressiveness
- Prolog: versatility and practical expressiveness of logic programming and the relational paradigm, formal (i.e., DCG) grammars as a programming feature, great utility for prototyping, potential benefits as a first programming language.

Bibliography

[1] P. Alevoor, P. Saeda & K. Kapoor, “On the decidability and matching issues for regex languages”, *Proc. Int. Conf. on Advances in Computing*, (Aswatha Kumar M., Selvarani R. & T. V. Suresh Kumar, eds.), Springer-Verlag, 2012, 137-145.

[2] American National Standards Institute, Inc., *IEEE Standard Pascal Computer Programming Language*, ANSI/IEEE X3.97-1983, 1983, 128 pp.

[3] M. A. Arbib, “Automaton automorphisms”, *Inform. and Control* 11 (1967), 147-154.

[4] W. F. Atchison, et al, “Curriculum 68: recommendations for academic programs in Computer Science”, *Comm. ACM* 11 (1968), 151-197.

[5] J. Backus, “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference”, *Proc. Int. Conf. on Information Processing*, UNESCO (1959), 125-132.

[6] J. Backus, “The History of FORTRAN I, II, and III”, *ACM SIGPLAN Notices* 13 (1978), 165-180.

[7] J. Backus, “Can programming be liberated from the von Neumann style? A Functional style and its algebra of programs”, *Comm. ACM* 21 (1978), 613-641.

[8] J. Backus, J. Williams & E. Wimmers, “An introduction to the programming language FL”, *Research Topics in Functional Programming* (D. A. Turner, ed.), Addison-Wesley, 1990, 219-247.

[9] T. P. Baker & A. C. Fleck, “Does Scope = Block in Pascal?”, *Pascal News* 17 (1980), 60-61.

[10] T. P. Baker & A. C. Fleck, “A Note on Pascal Scopes”, *Pascal News* 17 (1980), 62.

[11] B. H. Barnes, “Groups of automorphisms and sets of equivalence classes of inputs”, *Jour. ACM* 12 (1965), 561-565; also presented at ACM National Conf. (Denver, CO), 1963.

[12] Z. Bavel, “Structure and transition-preserving functions of finite automata”, *Jour. ACM* 15 (1968), 135-158.

[13] Z. Bavel, “On the number of automorphisms of a singly generated automaton”, *Comm. ACM* 13 (1970), 574-575.

[14] Z. Bavel, *Introduction to the Theory of Automata*, Reston Pub., 1983, 658 pp.

[15] E. Ben-Sasson, Y. kaplan, S. Kopparty, O. Meir & H. Stichtenoth, “Constant rate PCPs for circuit-SAT with sublinear query complexity”, *Jour. ACM* 63 (2016), 32:1-32:57.

[16] D. G. Bobrow, & B. Raphael, “A comparison of list-processing languages: including a detailed comparison of COMIT, IPL-V, LISP 1.5, and SLIP”, *Comm. ACM* 7 (1964), 231-240.

[17] C. Böhm & G. Jacopini, “Flow diagrams, Turing machines and languages with only two formation rules”, *Comm. ACM* 9 (1966), 366-371.

[18] J. M. Boyle & A. A. Grau, “An algorithmic semantics for ALGOL 60 identifier denotation”, *Jour. ACM* 17 (1970), 361-382.

[19] R. R. Brown, “Tape sets and automata”, *Jour. ACM* 11 (1964), 10-14.

[20] T. Budd, *A Little Smalltalk*, Addison-Wesley, 1987, 280 pp.

[21] M. Bullynck, “Programming primes (1968-1976)”, *History and Philosophy of Logic* 36 (2015), 229-241.

[22] C. Cˆampeanu, K. Salomaa, S. Yu, “A formal study of practical regular expressions”, *Int. Jour. Found. Comput. Sci*. 14 (2003), 1007–1018.

[23] C-H. Chang & H. Chang, “On the automorphism of reversible linear cellular automata”, *Information Sciences* 345 (2016), 217-225.

[24] N. Chomsky, “On certain formal properties of grammars”, *Inform. and Control* 2 (1959), 137-167.

[25] A. Church, “A set of postulates for the foundation of logic”, *Annals of Math*. 2 (1932-33), 33-34, 346-366, 839-864.

[26] A. Colmerauer & P. Roussel, “The birth of Prolog”, *History of Programming Languages – II* (T. J. Bergin Jr. & R. G. Gibson Jr., eds.), ACM, 1996, 331-367.

[27] O.-J. Dahl & K. Nygaard, “SIMULA – an ALGOL-based simulation language”, *Comm. ACM* 9 (1966), 671-678.

[28] O.-J. Dahl, B. Myhrhaug & K. Nygaard, “The SIMULA 67 common base language”, Norwegian Computing Center, Oslo, 1968.

[29] M. Davis, *Computability and Unsolvability*, McGraw-Hill, 1958, 210 pp.

[30] E. G. Daylight, “Dijkstra's Rallying Cry for Generalization: The Advent of the Recursive Procedure, Late 1950s–Early 1960s”, *Computer Journal* 54 (2011), 1756-1772.

[31] E. W. Dijkstra, “GOTO statement considered harmful”, *Comm. ACM* 11 (1968), 147-148.

[32] E. W. Dijkstra, "Notes on structured programming", EWD 249, Technical Univ., Eindhoven, Netherlands, 1969, 85 pp.

[33] A. D. Falkoff & K. E. Iverson, *APL\360: User's Manual*, IBM, 1968, 148 pp.

[34] A. D. Falkoff & K. E. Iverson, "The Evolution of APL", *ACM SIGPLAN Notices 13* (1978), 47-57.

[35] D. J. Farber, R. E. Griswold & I. P. Polonsky, "SNOBOL, A String Manipulation Language", *Jour. ACM* 11 (1964), 21-30.

[36] A. C. Fleck, Preservation of structure by certain classes of functions on automata and related group theoretic properties, Tech. Rpt. No. 16, Computer Laboratory, Michigan State University, May 1961, 28 pp.

[37] A. C. Fleck, "Structure preserving properties of certain classes of functions on automata", *Notices of Amer. Math. Society* 8:4 (1961), 582-17, p.340; and presented at the 1961 summer (Aug.) meeting of the Amer. Math. Society in Stillwater, OK.

[38] A. C. Fleck, "Isomorphism groups of automata", *Jour. ACM* 9 (1962), 469-476.

[39] A. C. Fleck, Algebraic structure of automata, Ph.D. thesis, Michigan State University Library, 108 286 THS, 1964, 66 pp.; *Dissertation Abstracts* 25, No.11 (9165), Order No, 65-678.

[40] A. C. Fleck, "On the automorphism group of an automaton", *Jour. ACM* 12 (1965), 566-569.

[41] A. C. Fleck, "On the strong connectedness of the direct product", *IEEE Trans. Elect. Comput.*, Vol. EC-16 (1967), 90.

[42] A. C. Fleck, "Towards a theory of data structures", *Jour. Computer and System Sciences* 5 (1971), 475-488.

[43] A. C. Fleck, “On the combinatorial complexity of context-free grammars”, *Information Processing* 71 (1972), North-Holland Pub., 59-60; also presented at IFIP71, Ljubljana, Yugoslavia.

[44] A. C. Fleck, S. T. Hedetniemi & R. H. Oehmke, “S-semigroups of automata”, *Jour. ACM* 19 (1972), 3-10,

[45] A. C. Fleck, “An analysis of grammars by their derivation sets”, *Inform. and Control* 24 (1974), 389-398.

[46] A. C. Fleck, “On the impossibility of content exchange through the by-name parameter transmission mechanism”, *ACM SIGPLAN Notices* 11 (1976), 38-41.

[47] A. C. Fleck, “Formal models for string patterns”, *Current Trends in Programming Methodology Vol. IV: Data Structuring* (R. Yeh, ed.), Prentice-Hall, 1978, 216- 240.

[48] A. C. Fleck, “Verifying abstract data types with SNOBOL4”, *Software – Practice and Experience* 12 (1982), 627-640.

[49] A. C. Fleck & R. S. Limaye, “Formal semantics and abstract properties of string pattern operations and extended formal language description mechanisms”, *SIAM Jour. Comput*. 12 (1983), 166-188.

[50] A. C. Fleck, "A proposal for the comparison of types in Pascal and associated semantic models", *Computer Languages* 9 (1984), 71-87.

[51] A. C. Fleck, "Structuring FP-style functional programs", *Computer Languages* 11 (1986), 55-63.

[52] A. C. Fleck, “A case study comparison of four declarative programming languages”, *Software—Practice and Experience* 20 (1990), 49-66.

[53] A. C. Fleck, *Formal Models of Computation*, World Scientific, 2001, 532 pp.

[54] A. C. Fleck, “Specifying and proving object-oriented programs”, *Proc. 2004 Hawaii Inter. Conf. on Computer Sciences*, 190-206.

[55] A. C. Fleck, “A simplified view of Nerode equivalence”, *Computing Letters* 1 (2005), 93-96.

[56] A. C. Fleck, “Prolog as the first programming language”, *ACM SIGSCE Bulletin* 39 (2007), 61-64.

[57] G. H. L. Fletcher, M. Gyssens, J. Faredaens & D. Van Gucht, “On the expressive power of the relational algebra on finite sets of relation pairs”, IEEE Trans. On Knowledge and Data Eng. 21 (2009), 939-942.

[58] R. W. Floyd, “*The syntax of programming languages – a survey*”, IEEE Trans. on Electronic Computers EC-13 (1964), 346-353.

[59] D. D. Freydenberger, “Extended regular expressions: succinctness and decidability”, *Theory of Computing Sys.* 53 (2013), 159-193.

[60] J. E. F. Friedl, *Mastering Regular Expressions* (3rd ed.), O’Reilly Media, Inc., 2006, 515 pp.

[61] F. Gécseg & I. Peák, *Algebraic Theory of Automata*, Akadémiai Kiadó, Budapest, 1972, 328 pp.

[62] S. R. Ghorpade & K. V. Kaipa, “Automorphism groups of Grassman codes”, *Finite Fields and Their Applications* 23 (2013), 80-102.

[63] S. Ginsberg, “On the reduction of superfluous states in sequential machines”, *Jour. ACM* 6 (1959), 259-282.

[64] S. Ginsburg, “Some remarks on abstract machines”, *Trans. Amer. Math Soc.* 96 (1960), 400-444.

[65] J. A. Goguen, J. W. Thatcher, E. G. Wagner & J. B. Wright, “Abstract data-types as initial algebras and correctness of data representations”, Proc. Conf. on Computer Graphics, Pattern Recognition and Data Structure, 1975, 89-93.

[66] J. A. Goguen, “Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs”, Proc. Conf. on Math. Studies of Infor. Proc. (E. Blum, M. Paul & S. Takasu, eds.), 1979, LNCS V.75, Springer-Verlag, 425-473.

[67] A. Goldberg & D. Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983, 714 pp.

[68] R. E. Griswold, J. F. Poage & I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, 1968, 221 pp.

[69] R. E. Griswold, J. F. Poage & I. P. Polonsky, *The SNOBOL4 Programming Language* (2nd ed.), Prentice-Hall, 1971, 258 pp.

[70] R. E. Griswold, “A history of the SNOBOL programming languages”, *History of Programming Languages*, ACM (1981), 601-645.

[71] J. W. Grzymala-Busse & Z. Bavel, “Characterization of state-independent automata”, Theor. Comput. Sci. 43 (1986), 1-10.

[72] J. V. Guttag, E. Horowitz & D. R. Musser, “The design of data type specifications”, ICSE ’76: Proc. 2nd Int. Conf. on Software Eng., IEEE, 1976, 414-430.

[73] J. V. Guttag, E. Horowitz & D. R. Musser, “Abstract data types and software validation”, *Comm. ACM* 21 (1978), 1048-1063.

[74] J. Hartmanis & R. E. Stearns, *Algebraic Theory of Sequential Machines*, Prentice-Hall, 1966, 211 pp.

[75] C. A. R. Hoare, “Record handling”, *ALGOL Bulletin* 21 (1965), 39-69.

[76] C. A. R. Hoare, “An Axiomatic approach to computer programming”, *Comm. ACM* 12 (1969), 576-580, 583.

[77] C. A. R. Hoare, “Proof of correctness of data representations”, *Acta Informatica* 1 (1972), 271-281.

[78] C. A. R. Hoare, “The Emperor’s old clothes”, *Comm. ACM* 24 (1981), 75-83.

[79] D. A. Huffman, “The synthesis of sequential switching circuits”, *Jour. Franklin Inst*. 257 (1954), 161-300.

[80] R. K. W. Hui, K. E. Iverson, E. E. McDonnell & A. T. Whitney, “APL\?”, *APL90 Conf. Proc., APL Quote-Quad* 20 (1990).

[81] IBM, *650 Magnetic Drum Data-Processing Machine Manual of Operation*, Form 22-6060-1, 1955, 111 pp.

[82] IBM, *Programmer’s Reference Manual (for the) Fortran Automatic Coding System for the IBM 704*, 1956, 51 pp.

[83] IBM, CPS Under TSO PRPO Specification, Prog. No. 5799-ADY, 1968, GH20-4315.

[84] M. Ito, “Generalized group-matrix type automata”, *Trans. Commun. Eng. Japan*, Sect. E59 (11) (1976), 9-13.

[85] K. E. Iverson, *A Programming Language*, John Wiley and Sons, Inc., 1962, 186 pp.

[86] K. Jensen & N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, 1974, 188 pp.

[87] R. T. Johnson & J. B. Morris, “Abstract data types in the MODEL programming language”, Proc. ACM Conf. on Data: Abstraction, Definition and Structure, *SIGPLAN Notices* 11 (1976), 36-46.

[88] Jsoftware Inc., *http://www.jsoftware.com*.

[89] J. R. Jump, “A note on the iterative decomposition of finite automata”, *Inform. and Control* 15 (1969), 424-435.

[90] A. C. Kay, “The early history of Smalltalk”, History of Programming Languages – II (T. J. Bergin Jr. & R. G. Gibson Jr., eds.), ACM, 1996, 511-598.

[91] S. C. Kleene, “Representation of events in nerve nets and finite automata”, *Automata Studies* (C. E. Shannon & J. McCarthy, eds.), Princeton Univ. Press (1956), 3-42.

[92] D. E. Knuth, “The remaining trouble spots in ALGOL 60”, *Comm. ACM* 10 (1967), 611-618.

[93] D. E. Knuth, “Semantics of context-free languages”, *Math. Sys. Theory* 2 (1968), 127-145; errata, ibid. 5 (1971), 95-96.

[94] D. E. Knuth, “Literate programming”, *The Computer Journal* 27 (1984), 97-111.

[95] K. B. Krohn & J. L. Rhodes, “Algebraic theory of machines”, *Proc. Symp. Math. Theory of Automata* (J. Fox, ed.), Polytechnic Press, 1963, 341-384.

[96] K. Krohn & J. Rhodes, “Algebraic theory of machines, I”, *Trans. Amer. Math. Soc*. 116 (1965), 450-464.

[97] J. A. N. Lee, “Computer Pioneers”, IEEE Computer Society, 1995, http://history.computer.org/pioneers/weeg.html.

[98] B. Liskov & S. Zilles, “Programming with abstract data types”, *ACM SIGPLAN Notices* 9 (1974), 50-59.

[99] B. Liskov & S. Zilles “Specification techniques for data abstractions”, Int. Conf. on Reliable Software, *ACM SIGPLAN Notices* 10 (1975), 72-87.

[100] K. C. Liu & A. C. Fleck “String pattern matching in polynomial time”, Proc. 6th ACM Symp. on Principles of Prog. Lang. (POPL ’79), ACM, 222-225.

[101] R. L. London, “Who Earned First Computer Science Ph.D.?”, BLOG@CACM (2013), https://cacm.acm.org/blogs/blog-cacm/159591-who-earned-first-computer-science-ph-d/fulltext

[102] Z. Manna & J. Vuillemin, “Fixpoint approach to the theory of computation”, *Comm. ACM* 15 (1972), 528-536.

[103] Y. Masunaga, S. Noguchi & J. Oizumi, “A characterization of automata and a direct product decomposition”, *Jour. Comput. and Sys. Sci*. 13 (1976), 74-89.

[104] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, Part I”, *Comm. ACM* 3, 4 (1960), 184-195.

[105] W. McCulloch & W. Pitts, “A logical calculus of the ideas in nervous activity”, *Bull. of Math. Biophysics* 5 (1943), 115-133.

[106] G. H. Mealy, “A method for synthesizing sequential circuits”, *Bell Systems Tech. Jour.*, V. 34 (1955), 1045-1079.

[107] E. F. Moore, “Gedanken-experiments on sequential machines”, *Automata Studies*, Princeton (1956), 129-153.

[108] H. L. Morgan & R. A. Wagner, “PL/C: the design of a high-performance compiler for PL/I”, *Proc. Spring Joint Comput. Conf*. (AFIPS ’71), 503-510.

[109] P. Naur (ed.), et al., “Report on the Algorithmic Language ALGOL 60”, *Comm. ACM* 3, 5 (1960), 299-314.

[110] P. Naur (ed.), et al., “Revised Report on the Algorithmic Language ALGOL 60”, *Comm. ACM* 6,1 (1963), 1-17.

[111] P. Naur, “The European side of the last phase of the development of ALGOL 60”, *History of Programming Languages* (I. R. Wexelblat, ed.), ACM (1981), 92-139.

[112] A. Newell, & F. M. Tonge, “An Introduction to Information Processing Language V”, *Comm. ACM* 3 (1960), 205-211.

[113] K. V. Nori et al, “Pascal-P implementation notes”, *Pascal – the Language and its Implementation* (D. W. Barron, ed.), John Wiley & Sons, 1981, 125-170.

[114] M. J. O’Donnell, Computing in Systems Described by Equations, Lect. Notes in Comp. Sci., Vol. 58 (1977), Springer-Verlag, 111 pp.

[115] R. H. Oehmke, “On the structures of an automaton and its input semigroup”, *Jour. ACM* 10 (1963), 521-525.

[116] M. O. Rabin & D. Scott, “Finite automata and their decision problems”, *IBM Jour. of Research and Development* 3 (1959), 114-125.

[117] G. Radin & H. P. Rogoway, “NPL: Highlights of a New Programming Language”, *Comm. ACM* 8 (1965), 9-17.

[118] B. Randell & L. J. Russell, *ALGOL 60 Implementation*, Academic Press, 1964, 418 pp.

[119] J. E. Robertson, “The ORDVAC and the ILLIAC”, *A History of Computing in the Twentieth Century* (N. Metropolis, J. Howlett & G-C. Rota, eds.), Academic Press, Inc., 1980, 347-364.

[120] J. A. Robinson, “A machine-oriented logic based on the resolution principle”, *Jour. ACM* 12 (1965), 23-41.

[121] S. Sataluri & A. C. Fleck “Semantic specification using logic programs”, Logic Programming: Proc. North Amer. Conf. (E. L. Lusk & R. A. Overbeek, eds.), V. 2 (1989), MIT Press, 772-794.

[122] P. Sebastian & T. P. Johnson, “Automorphism group of an inverse fuzzy automaton”, *Annals of Pure and Applied Math*. 2 (2012), 67-73.

[123] C. E. Shannon, “A symbolic analysis of relay and switching circuits”, *Electrical Eng*. 57 (1938), 713-723.

[124] P. W. Shantz, R. A. German, J. G. Mitchell, R. S. K. Shirley & C. R. Zarnke, “WATFOR–The University of Waterloo FORTRAN IV Compiler”, *Comm. ACM* 10 (1967), 41-44.

[125] D. C. Spriestersbach, *The Way It Was: The University of Iowa*, 1964-1989, University of Iowa Press, 1999, 263 pp.

[126] I. E. Sutherland, “Sketchpad: a man-machine graphical communication system”, AFIPS ‘63 – Proc. Spring Joint Computer Conf., 1963, ACM, 329-346.

[127] K. Thompson, “Regular expression search algorithm”, *Comm. ACM* 11 (1968), 419-422.

[128] J. Tian, X. Zhao & Y. Shao, “On structure and representations of cyclic automata”, *Theoretical Comput. Sci*. 609 (2016), 344-360.

[129] J. Tiuryn, “Some results on the decompositiion of finite automata”, *Inform. and Control* 38 (1978), 288-297.

[130] C. A. Trauth, “Group-type automata”, *Jour. ACM* 13 (1966), 170-175.

[131] A. M. Turing, “On computable numbers, with an application to the Enscheidungsproblem”, *Proc. London Math*. Soc., Series 2, V. 42 (1936), 230-265; correction, ibid. V. 43 (1937), 544-546.

[132] D. A. Turner, “A new implementation technique for applicative languages”, *Software –Practice and Experience* 9 (1979), 31-49.

[133] D. A. Turner, “Miranda: a non-strict functional language with polymorphic types”, *Proc. IFIP Int. Conf. on Functional Programming Languages and Computer Architecture*, Lect. Notes in Comp. Sci., V. 201 (1985), Springer-Verlag, 1-16.

[134] K. Uemura, “Semigroups and automorphism groups of strongly connected automata”, *Math. Sys. Theory* 8 (1974), 8-14.

[135] USA Standards Institute, Inc., *USA Standard FORTRAN*, ANSI X3.9-1966, 1966, 36 pp.

[136] J. von Neumann, First draft of a report on the EDVAC, Moore School of Elect. Eng., Univ. of Pennsylvania, 1945, 101 pp.

[137] G. P. Weeg, Some group theoretic properties of strongly connected automata, Tech. Rpt., Computer Laboratory, Michigan State University, May 1961.

[138] G. P. Weeg, “The structure of an automaton and its operation preserving transformation group”, *Jour. ACM* 9 (1962), 345-349.

[139] G. P. Weeg, “The group and semigroup associated with automata”, *Proc. Symp. Math. Theory of Automata* (J. Fox, ed.), Polytechnic Press, 1963, 257-266.

[140] G. P. Weeg, “The structure of the semigroup associated with automata”, *Computer and Information Science* (J. T. Tou & R. H. Wilcox, eds.; symposium proceedings, Northwestern Univ., June 1963), Spartan Books Inc., 1964, 230-245.

[141] G. P. Weeg, "The automorphism group of the direct product of strongly related automata", *Jour. ACM* 12 (1965), 187-195; correction: ibid. 14 (1967), 421.

[142] G. P. Weeg & G. B. Reed, *Introduction to Numerical Analysis*, Blaisdell Pub. Co., 1966, 184 pp.

[143] Wikipedia, "List of programming languages", https://en.wikipedia.org/wiki/List_of_programming_languages

[144] N. Wirth & H. Weber, "EULER: a generalization of ALGOL, and its formal definition: Part I", *Comm. ACM* 9, 1 (1966) 13-25; "Part II", ibid. 9, 2 (1966), 89-99.

[145] N. Wirth & C. A. R. Hoare, "A contribution to the development of ALGOL", *Comm. ACM* 9 (1966), 423-432.

[146] N. Wirth, "The programming language Pascal", *Acta Informatica* 1 (1971), 35-63.

[147] N. Wirth, "Recollections about the development of Pascal", *History of Programming Languages – II* (T. J. Bergin Jr. & R. G. Gibson Jr. eds.), 1996, ACM, 97-120.

[148] W. A. Wulf, R. L. London & M. Shaw, "An introduction to the construction and verification of Alphard programs", *IEEE Trans. on Software Eng* SE-2 (1976), 253-265.

[149] N. A. Zafar, A. Hussain & A. Ali, "Verifying monoid and group morphisms over strongly connected algebraic automata", *Jour. Software Eng. & Appl*. 3 (2010), 803-812.